

AD-A230 751

AFOSR-TR-

199

The Synthesis of Intelligent Real-Time Systems
Final Technical Report
Covering Period April 1989 to November 1990

Stanley J. Rosenschein
Leslie Pack Kaelbling

Technical Report No. TR/90-03

November 9, 1990

TELEOS
RESEARCH

DTIC
ELECTE
JAN 09 1991
S B D

2

The Synthesis of Intelligent Real-Time Systems
Final Technical Report
Covering Period April 1989 to November 1990

Stanley J. Rosenschein
Leslie Pack Kaelbling

Technical Report No. TR/90-03

November 9, 1990

Prepared by:
Teleos Research
576 Middlefield Road
Palo Alto, CA 94301

Prepared for:
Air Force Office of Scientific Research
Bolling Air Force Base
DC 20332-6448

DTIC
ELECTE
S B D
JAN 09 1991

Contract Number F49620-89-C-0055

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

REPORT DOCUMENTATION PAGE

Form Approved
GSA No. 0704-0188

Public reporting burden for this document of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Nov. 9, 1990	3. REPORT TYPE AND DATE Contractor Report: Final JVERED Apr. 89-Nov. 90
4. TITLE AND SUBTITLE The Synthesis of Intelligent Real-Time Systems.		5. FUNDING NUMBERS F49620-89-0055 Project No. 2304/A7 2304/A2	
6. AUTHOR(S) Stanley J. Rosenschein and Leslie P. Kaelbling		7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Teleos Research 576 Middlefield Road Palo Alto, CA 94301	
8. PERFORMING ORGANIZATION REPORT NUMBER TR/90-03		9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DCASMA - Monitoring AFOSR (Sponsoring) Bldg. 41Q, Bolling Air Force Base Washington, D.C. 20332-6448	
10. SPONSORING/MONITORING AGENCY REPORT NUMBER F49620-89-C-0055 (PII No.)		11. SUPPLEMENTARY NOTES Stanley J. Rosenschein, Tel: 415/328-8803 AFOSR Program Manager: Dr. Abraham Waksman, Tel: 202/767-5025	
12a. DISTRIBUTION/AVAILABILITY STATEMENT UNCLASSIFIED/UNLIMITED		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Teleos Research, under the sponsorship of the Air Force Office of Scientific Research, has carried out a two-year program of research on "The Synthesis of Intelligent Real-Time Systems." The purpose of the effort was to develop and extend theories and techniques that facilitate the design and implementation of intelligent real-time systems. In particular, Teleos has extended situated-automata theory to apply to situations in which the system has probabilistic information about the world; designed and built a high-level, declarative programming tool for synthesizing efficient programs that track dynamic conditions in the world; clarified the theoretical relationships between Gapps, an existing declarative programming tool for describing action strategies, and the newly designed tool; investigated the possibility of moving the burden of developing correct programs from the human programmer to the agent itself through the use of algorithms that allow the agent to learn from trial and error; applied the principles of situated-automata theory to the understanding of existing vision algorithms and the development of new ones; and tested these theoretical principles and design tools in a real robotic domain.			
14. SUBJECT TERMS REAL-TIME SYSTEMS; REACTIVE CONTROL; REINFORCEMENT LEARNING; REAL-TIME PERCEPTION		15. NUMBER OF PAGES 173	
16. PRICE CODE		17. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	
18. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		19. LIMITATION OF ABSTRACT UNCLASSIFIED	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

1 Objectives of the Research Effort

Teleos Research, under the sponsorship of the Air Force Office of Scientific Research, has carried out a two-year program of research on "The Synthesis of Intelligent Real-Time Systems." The purpose of the effort was to develop and extend theories and techniques that facilitate the design and implementation of intelligent real-time systems. These are embedded computer systems that are linked to the external world through sensors and effectors and are programmed to interpret sensor data and to produce flexible, goal-directed behavior continuously and in real time. Systems of this kind will be of crucial importance to a wide variety of military and industrial applications, including robotics, process control, real-time situation monitoring, and space applications. If this potential is to be realized, better techniques will be required for producing the sophisticated software that lies at the heart of such systems.

In previous research, Teleos personnel have developed situated automata theory, a new approach toward modeling and programming intelligent real-time systems. This approach combines the flexibility of symbolic reasoning systems with the performance of real-time control systems. By identifying and encapsulating high-level abstractions, it is possible to raise the conceptual level at which such systems are programmed and to improve the efficiency of the programmer, as well as the capabilities of the target system.

The objectives of the research effort carried out for the AFOSR were to test and extend this existing approach in the following ways:

- Extend situated-automata theory to apply to situations in which the system has probabilistic information about the world;
- Design and build a high-level, declarative programming tool for synthesizing efficient programs that track dynamic conditions in the world;
- Understand the theoretical relationships between Gapps, an existing declarative programming tool for describing action strategies, and the newly designed tool;
- Explore the possibility of moving the burden of developing correct programs from the human programmer to the agent itself through the use of algorithms that allow the agent to learn from trial and error;
- Apply the principles of situated-automata theory to the understanding of existing vision algorithms and the development of new ones; and
- Test theoretical principles and design tools in a real robotic domain.

The combined progress in each of these areas has allowed us to take significant steps toward an ideal situation in which (1) a programmer gives a high-level, declarative specification of the dynamics of the world and of the intended behavior of the agent; (2) the specification is compiled into highly efficient code for a real robotic agent with complex effectors and visual sensors; and (3) the agent acts in the real world, filling out the details and correcting errors in its specification by learning from its experience in the real world.

2 Status of the Research Effort

In this section we will summarize our progress in each of the areas described above and provide references to more detailed accounts, which have been included as appendices to this report.

- We finished the design and implementation of Ruler, a declarative language from which efficient declarative programs can be synthesized. Ruler allows the user to specify the behavior of the world using Prolog-like rules. The user then selects particular properties of the world that should be tracked and the compiler, using backward-chaining proof techniques, generates a circuit that tracks the desired properties in the world. This system and its theoretical foundations are discussed in detail in Appendix A.
- Stanley Rosenschein and Michal Irani (a visiting student) developed a probabilistic formulation of situated-automata theory, including a calculus for the combination of probabilistic information. In the deterministic version of situated-automata theory, the information content of an agent's state is modeled in terms of environment conditions *implied* by the agent's being in that state. The theory can be generalized—and its utility extended—by allowing situations in which the correlation between the agent's state and the environment is *probabilistic* rather than deterministic. Unfortunately, because of the non-monotonic nature of probabilities, this extension does not immediately yield a compositional design methodology; a condition which is highly probable given the state of a component may be highly improbable given the states of other components. To overcome this problem, we developed an approach based on *stable probabilities* where the designer makes assertions that bound the probability of certain propositions given partial information about the agent's state. A prototype design tool was implemented in Prolog that, like Ruler, derived circuits compositionally from declarative specifications, but unlike Ruler was grounded in the more general probabilistic model of information.
- There is an elegant theoretical relationship between the notion of a "goal," as used in the Gapps language, and the notion of "information," as used in the Ruler language. If an agent can be thought of as having a single overall goal, N , then for it to have another goal, P , can be modeled as the agent's having the information that P implies N . This duality of goals and information is the subject of a more detailed discussion presented in Appendix B.

In addition to this theoretical work, the Rex and Gapps languages were extended and enhanced in a variety of ways. In order to support robotic experimentation and more efficient debugging, new code generators that generate Lisp and C as output were added to the Rex compiler. The Gapps compilation process was sped up significantly through the addition of a caching mechanism. Finally, a new construct was added to the Rex language that allows the execution of individual program modules to be triggered by other conditions in the program. This construct increases the efficiency of programs that have modules whose results are used only occasionally.

- The topic of reinforcement learning, or learning from trial and error, was studied extensively by Leslie Kaelbling. The results of this study included a number of new algorithms for efficient reinforcement learning in embedded agents and culminated in a demonstration of these techniques on a small mobile robot. The foundations of this approach to learning are described in Appendix C and some algorithms for efficiently learning Boolean functions in k -DNF from reinforcement are described in Appendix D. In addition, Kaelbling's Ph.D. thesis on this topic is included under separate cover.
 - We studied a variety of existing algorithms for machine vision. In particular, we used situated-automata-theoretic techniques to characterize several model-based vision algorithms, including Goad's method and Grimson's method, both of which match model features to image features to generate and filter hypotheses about object identity and to refine information about object parameters. While these investigations yielded a better understanding of the logical basis of these particular algorithms, they did not immediately suggest how to generalize the algorithms to less constrained domains nor how to compile declarative descriptions of such domains into perceptual recognition circuitry. These remain important topics for future research.
- In addition, David Chapman extended his work on the design of a set of visual primitives and routines over those primitives that can be used to support reactive behavior in embedded agents. Appendix E describes his work in detail.
- Finally, we designed an architecture for a complex robotic demonstration system. The specification for the architecture, included as Appendix F, includes descriptions of a class of demonstration tasks, an efficient symbolic database and query language, and an interface to low-level machine-vision tools. A large part of the specification was implemented, and the rest remains to be carried out under future research projects.



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

3 Publications

- H. Keith Nishihara, "Tests of a Sign Correlation Model for Binocular Stereo," *Investigative Ophthalmology and Visual Science*, vol. 30, no. 3, March, 1989.
- H. Keith Nishihara, "Psychophysical and Computational Tests Comparing the Sign-Correlation and Zero-Crossing Models of Human Stereo Vision," *Image Understanding and Machine Vision, 1989 Technical Digest Series, 14*, Optical Society of America, Washington, D.C., 1989.
- Stanley J. Rosenschein, "Synthesizing Information-Tracking Automata from Environment Descriptions," in *Proceedings of the First Annual Conference on Principles of Knowledge Representation*, Toronto, Canada, May, 1989.
- Stanley J. Rosenschein, "Synthesizing Information-Update Functions Using Off-Line Symbolic Processing," in *Proceedings of the Society of Photo-Optical Instrumentation Engineers Symposium on Advances in Intelligent Robotics Systems*, Philadelphia, Pennsylvania, 1989.
- Stanley J. Rosenschein and Leslie Pack Kaelbling, "Integrating Planning and Reactive Control," in *Proceedings of the NASA/JPL Space Telerobotics Conference*, Pasadena, California, 1989.
- Leslie Pack Kaelbling, "A Formal Framework for Learning in Embedded Systems," in *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, New York, 1989.
- Leslie Pack Kaelbling, *Learning in Embedded Systems*, Ph.D. Dissertation, Stanford University, 1990. Also published as Teleos Research Technical Report No. TR-90-04, August, 1990.
- Leslie Pack Kaelbling and Stanley J. Rosenschein, "Action and Planning in Embedded Agents," in *Robotics and Autonomous Systems*, vol. 6, pp. 35-48, 1990. Also in *New Architectures for Autonomous Agents: Task-Level Decomposition and Emergent Functionality*, P. Maes, Ed., MIT Press (in press).
- Leslie Pack Kaelbling, "Learning Functions in k -DNF from Reinforcement," in *Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, June, 1990.
- David Chapman, *Intermediate Vision: Architecture, Implementation, and Use*, Teleos Research Technical Report No. TR-90-06, October, 1990. Submitted to *Cognition*.
- Leslie Pack Kaelbling, "Generating Complex Behavior for Computer Agents," in *Proceedings of the DARPA Planning Workshop*, San Diego, California, November, 1990.

- Leslie Pack Kaelbling, "Foundations of Learning in Autonomous Agents," in *Robotics and Autonomous Systems* (in press). Also in *Toward Learning Robots*, W. Van de Velde, Ed., MIT Press (in press).

4 Personnel

Supervisory:

- Stanley J. Rosenschein, Ph.D., 1975: "Structuring a Pattern Space, With Applications to Lexical Information and Event Interpretation."

Senior Professional:

- H. Keith Nishihara, Ph.D., 1978: "Representation of the Spatial Organization of Three-Dimensional Shapes for Visual Recognition."

Professional:

- David Chapman, Ph.D., 1990: "Vision, Instruction, and Action."
- Neil Hunt, Ph.D., 1989: "Tools for Image Processing and Computer Vision."
- Leslie Pack Kaelbling, Ph.D., 1990: "Learning in Embedded Systems."
- Jeffrey R. Kerr, Ph.D., 1985: "An Analysis of Multi-Fingered Hands."
- Nathan J. Wilson, M.A., 1987: "Developing a Computational Model of Biological Motion to Study Concept Formation"

5 Interactions

5.1 Papers Presented

- H. Keith Nishihara, "Tests of a Sign Correlation Model for Binocular Stereo," *Investigative Ophthalmology and Visual Science*, vol. 30, no. 3, March, 1989.
- H. Keith Nishihara, "Psychophysical and Computational Tests Comparing the Sign-Correlation and Zero-Crossing Models of Human Stereo Vision," *Image Understanding and Machine Vision, 1989 Technical Digest Series, 14*, Optical Society of America, Washington, D.C., 1989.
- Leslie Pack Kaelbling, "Foundations of Learning in Autonomous Agents," at the Workshop on Representation and Learning in Autonomous Agents, Lagos, Portugal, 1988.
- Stanley J. Rosenschein, "Synthesizing Information-Tracking Automata from Environment Descriptions," at the First Annual Conference on Principles of Knowledge Representation, Toronto, Canada, May, 1989.
- Stanley J. Rosenschein, "Synthesizing Information-Update Functions Using Off-Line Symbolic Processing," at the Society of Photo-Optical Instrumentation Engineers Symposium on Advances in Intelligent Robotics Systems, Philadelphia, Pennsylvania, 1989.
- Stanley J. Rosenschein and Leslie Pack Kaelbling, "Integrating Planning and Reactive Control," at the NASA/JPL Space Telerobotics Conference, Pasadena, California, 1989.
- Leslie Pack Kaelbling, "A Formal Framework for Learning in Embedded Systems," at the Sixth International Workshop on Machine Learning, Ithaca, New York, 1989.
- Leslie Pack Kaelbling, "Learning Functions in k -DNF from Reinforcement," at the Seventh International Conference on Machine Learning, Austin, Texas, June, 1990.
- Leslie Pack Kaelbling, "Generating Complex Behavior for Computer Agents," at the DARPA Planning Workshop, San Diego, California, October, 1990.

5.2 Other Presentations

- Leslie Pack Kaelbling, "Intelligent Robots in the Real World," invited talk at the Eleventh IFIP World Computer Congress, San Francisco, California, 1989.
- H. Keith Nishihara, "A Machine Theory for Human Stereo Vision," Apple Computer, August, 1989.
- Stanley J. Rosenschein, participant in NASA/Ames Applications Workshop, Moffett Field, California, October, 1989.

- Stanley J. Rosenschein, guest speaker at Artificial Intelligence/Robotics Seminar Series, Computer Science Division, University of California, Berkeley, California, November, 1989.
- Stanley J. Rosenschein and Leslie Pack Kaelbling, participants in Workshop on Intelligent Real-Time Problem Solving, Santa Cruz, California, November 1989.
- Stanley J. Rosenschein, co-organizer and presenter with M. Pollack and M. Bratman, seminar series, "Models of Rational Agency," for Center for the Study of Language and Information, Stanford University, Stanford, California. Fall, 1989.
- H. Keith Nishihara, "Hidden Information in Transparent Stereograms," CCRMA Seminar, Stanford University, March, 1990.
- Leslie Pack Kaelbling, "Planning and Action in Robotics and AI," invited talk at the International Symposium: Artificial Intelligence, What Reality?, Rabat, Morocco, May, 1990.
- David Chapman, Leslie Pack Kaelbling, and Stanley J. Rosenschein, participants in DARPA Workshop on Benchmarks and Metrics for Integrated Agent Architectures, July, 1990.
- Stanley J. Rosenschein, "Reasoning and Acting in Real Time," invited talk at the Eighth National Conference on Artificial Intelligence, Boston, Massachusetts, August, 1990.

6 Discoveries and Inventions

There have been no new discoveries, inventions, or patent disclosures or applications stemming from this research effort.

7 Other Statements

The following papers, which are included as appendices to this report, provide a detailed description of the research progress achieved under this contract:

- Stanley J. Rosenschein, "Synthesizing Information-Tracking Automata from Environment Descriptions."
- Leslie Pack Kaelbling and Stanley J. Rosenschein, "Action and Planning in Embedded Agents."
- Leslie Pack Kaelbling, "Foundations of Learning in Autonomous Agents."
- Leslie Pack Kaelbling, "Learning Functions in k -DNF from Reinforcement."
- David Chapman, "Intermediate Vision: Architecture, Implementation, and Use."
- Leslie Pack Kaelbling, Neil D. Hunt, Stanley J. Rosenschein, H. Keith Nishihara, Nathan J. Wilson, Laura E. Wasylenki, and Jeffrey R. Kerr, "Cooperative Robot Demonstration: Working Document"

In addition, Leslie Pack Kaelbling completed her Ph.D. thesis under the partial sponsorship of this contract. It is titled *Learning in Embedded Systems*, and has been included under separate cover.

A Synthesizing Information-Tracking Automata from Environment Descriptions

Synthesizing Information-Tracking Automata from Environment Descriptions

Stanley J. Rosenschein

Teleos Research

Technical Report No. 2

July 3, 1989

Abstract

This paper explores the synthesis of finite automata that dynamically track conditions in their environment. We propose an approach in which a description of the automaton is derived automatically from a high-level declarative specification of the automaton's environment and the conditions to be tracked. The output of the synthesis process is the description of a sequential circuit that at each clock cycle updates the automaton's internal state in constant time, preserving as an invariant the correspondence between the state of the machine and conditions in the environment. The proposed approach allows much of the expressive power of declarative programming to be retained while insuring the reactivity of the run-time system.

This work was supported in part by a gift from the System Development Foundation.

Synthesizing Information-Tracking Automata from Environment Descriptions

1 Introduction

This paper is concerned with the synthesis of finite automata whose internal states are provably correlated with changing conditions in the environment. In earlier work [Rosenschein1985, Rosenschein and Kaelbling1986], we investigated the mathematical foundations of embedded machines and direct methods of programming them. Later research was aimed at raising the conceptual level of the programming task by exploring declarative techniques for synthesizing their action-selection circuitry [Kaelbling1988]. In this paper, we extend this line of research to perceptual updates, that is, the computations responsible for updating the internal information state of the machine. We present techniques that allow programmers to describe the environment in which a machine is to be embedded along with conditions to be tracked and to have these descriptions algorithmically transformed into provably real-time circuitry for tracking those conditions in the environment. Information about these conditions would be used by other parts of the system to guide action.

Mainstream theoretical AI has developed models of information and action based on formalized commonsense psychology. In this approach, intelligent computer systems are modeled as having at their disposal a set of propositional "beliefs," usually assumed to be embodied in a set of symbolic expressions, such as logical formulas, whose intended semantics are clear to the designer. Some of these beliefs are provided by the designer as part of a knowledge base, while others are produced by the perceptual system at run time. In addition, the system contains inference procedures for dynamically deriving new beliefs from old and for continuously revising beliefs over time in response to sensory inputs (and perhaps reflection.) In this way, the designer can arrange for the agent to have access to a much more complex set of beliefs than could have been enumerated explicitly in advance. The designer also provides symbolic representations of the goals the agent is to pursue. The agent continuously attempts to deduce which actions it should take to achieve its goals and then performs those actions.

By modeling the information available to the system as symbolic facts deducible

by the system, the traditional approach allows the methods of symbolic logic, including automated symbolic inference, to be applied to problems in agent design. Of particular importance is the availability of a clear semantics for non-numerical data structures that are used to represent qualitative information about the world. These are attractive features—ones we would like to preserve. However, the traditional AI approach also has some other, less attractive, features which we hope to eliminate. For example, in applications requiring continuous, high-speed interaction with the environment, the computational cost of formally deriving facts from a data base of logical premises and of keeping the data base consistent with the world is often prohibitive. This has been a severe obstacle to building high-performance embedded computer systems based on the model of the intelligent agent as symbolic reasoner.

Situated-automata theory is a framework for reconciling the attractive features of AI methods (non-numerical descriptions of the world) and of control-theoretic methods (continuous constant-time updating of internal representations and guaranteed response.) The central observation of situated-automata theory is this: It is not the run-time symbols or numbers, per se, that are of significance, it is the fact that (1) they are semantically meaningful to the designer, that is, they stand for well-defined world conditions, and (2) the machine is designed in such a way that the world condition represented by the value of an internal location will indeed hold when that location has that value.

In this paper, we apply the situated-automata framework to the problem of synthesizing machines that track semantically complex conditions in the environment using constant-time update circuitry. We describe how inference techniques can be used at compile time to carry out the synthesis automatically, given symbolic descriptions of the environment and of the information to be tracked.

2 Basic concepts: A model of information

The mathematical framework of situated-automata theory takes as its starting point a model of dynamic systems. Consider a physical or computational system consisting of a set of locations that can be in different states over time. These states can be thought of as actual physical states or as abstract data values that might be stored in the register of a computer. Let T be a set of times, L a set of locations, and let each location a take on values from some set, D_a , with compound locations $[a, b]$ taking on values in $D_a \times D_b$. Let the union of all value domains be designated by D . Each possible "trajectory" of values can be given by a function $w : L \times T \rightarrow D$, in which $w(a, t)$ is the value of location a at time t in trajectory w . Following the terminology of possible-worlds semantics, we call these trajectories "worlds."

In physical or computational systems that operate according to fixed rules or constraints, not every world is consistent with the laws of nature. This can be

captured mathematically by identifying some designated subset of worlds that are consistent with the intended constraints. We shall call this set of possible worlds W . Let Φ , the set of *propositions* or *world conditions*, be the set of all subsets of $W \times T$. Intuitively, a condition $\varphi \in \Phi$ corresponds to the set of world-time points at which that condition holds. We sometimes write $\varphi(w, t)$ rather than $\langle w, t \rangle \in \varphi$ when we wish to assert that the condition φ holds at w, t .

By definition, Φ has the structure of a Boolean algebra of sets: a condition φ can *imply* (be a subset of) another condition ψ , we can take the *meet*, $\varphi \cap \psi$, of two conditions, and so on. Furthermore, the structure of Φ allows us to define two mathematical objects useful for characterizing world dynamics: the *initial condition* $\varphi_0 = \{\langle w, 0 \rangle \mid w \in W\}$ and the *strongest postcondition function* $S : \Phi \rightarrow \Phi$, where $S(\varphi) = \{\langle w, t+1 \rangle \mid \varphi(w, t)\}$. The initial condition φ_0 and the strongest postcondition function S will be used later to characterize machine synthesis.

The restriction on what is possible gives rise directly to a notion of information. The information contained in a location's value is modeled as the strongest proposition consistent with that location's having that value. Formally stated, to every location (or compound location) a , we associate a function, $M_a : D_a \rightarrow \Phi$ that maps a 's values into propositions. This function is defined as follows: $M_a(v) = \{\langle w, t \rangle \mid w(a, t) = v\}$. To say that a location a has the information that φ in world w at time t is to say $M_a(v)$ implies φ , in other words, that the proposition φ is true at each world and time in which a has the same value it has in world w at time t .

As defined, the concept of information is very abstract, representing the totality of what must be the case, given that some location in the machine has the value it does. For this notion to be of practical use, we must find ways of expressing in understandable terms particular, more limited, aspects of this total information content. This is the proper role of logic. By defining logical languages whose formulas express propositions of interest, we can conveniently describe the content of propositions included in an agent's information state, such as $\text{in}(\text{book}, \text{room1}) \vee \text{in}(\text{book}, \text{room2})$. Furthermore, modal logics of knowledge can be used to assert facts about the information relation itself, such as whether particular locations have or do not have particular information, e.g., $\neg K(a, \text{in}(\text{book}, \text{room2})) \wedge K(b, \text{in}(\text{book}, \text{room2}))$, which asserts that location a does not contain the information that the book is in room 2, while location b does. These logics are explored more fully in [Rosenschein and Kaelbling1986] and [Halpern and Moses1985]. In this paper, we will use letters p, q, \dots and standard logical operators \wedge, \vee, \dots in formulas that express the information carried in a location's value.

When we wish to consider machines with very large state sets, we regard the machine as being constructed from a network of components, with the state set of the whole machine corresponding to the Cartesian product of the state set of the individual components. Fortunately, there are straightforward techniques for inferring informational properties of aggregates from informational properties of

their components. For instance, the following can easily be shown to be valid:

$$M_{[a,b]}([u,v]) = M_a(u) \cap M_b(v)$$

We refer to this property as *spatial monotonicity*. It follows that if location a carries the information that p holds and location b carries the information that q holds, then the aggregate location $[a,b]$ carries the information that the conjunctive condition $p \wedge q$ holds. Spatial monotonicity is useful in synthesis because it means that subsystems can be developed independently and composed in a meaningful way.

It is important to observe that location a can carry information about p without explicitly encoding a symbolic formula representing p ; any value of the location that is systematically correlated with p will suffice. Different locations might have different states representing the same proposition p , and the same data values might have different informational significance at different locations. In general, an infinite number of formulas will follow from the information contained in a finite value, but since the formulas need not be separately represented, this causes no problem. Indeed, the computational complexity of updating a location's value so that it tracks changing conditions in the environment is entirely decoupled from the computational complexity of the symbolic inference problem for the logical language that expresses the conditions being tracked. This fact is crucial for understanding how seemingly complex semantic conditions can be monitored in real time.

These considerations lead directly to certain abstractions useful for describing how information is represented in machine states and how it is re-represented as it moves from location to location within a machine. We call these abstractions *informational data types*.

3 Informational data types

Recall that the function M_a maps a 's values (elements of D_a) to the abstract propositions with which they are correlated. As such, we can think of it as a "meaning" function. It is also useful to consider an inverse to these meaning functions, namely, "representation" functions that map propositions back to the data values that encode them. Let us define an *informational data type* to be a triple $\tau = \langle D, M, R \rangle$, with D being a set of data values, M a meaning function from D to Φ , and R a representation function from Φ to D . Intuitively, if a location is of type τ , then whenever it takes on the value $v \in D$, the world is intended to satisfy condition $M(v)$, and if the world satisfies condition φ , it is appropriate for the location to take on the value $R(\varphi)$. These two functions must satisfy the property that φ implies $M(R(\varphi))$ for all $\varphi \in \Phi$, that is, the representation map must be consistent with the meaning map. Since the implication is only one way, the representation of a world condition in the machine will, in general, not be information preserving. An

extreme case of this is when $M(R(\varphi)) = \text{true}$ and contains no information at all. We generally assume $R(\varphi)$ to be maximally specific, that is if $R(\varphi) = v_i$, then there is no v_j such that $M(v_j)$ implies, but is not implied by, $M(v_i)$.

Informational data types provide a way of analyzing the localization of information in a machine, including the computational complexity of such localization. Given two informational data types, $\tau_1 = \langle D_1, M_1, R_1 \rangle$ and $\tau_2 = \langle D_2, M_2, R_2 \rangle$, we can define a translation function that "re-represents" the content implicit in the values of a location of type τ_1 in the language of τ_2 . Mathematically, the translation function is a mapping $T_2^1 : D_1 \rightarrow D_2$ that is defined as follows: $T_2^1(v) = R_2(M_1(v))$, i.e., the representation, in the second "language," of the meaning, in the first "language," of v . The computability and complexity of these translations remain to be determined and are greatly affected by the choice of representation, that is, by the specific nature of M and R and not merely by the range of propositions encoded.

Although the informational concepts developed thus far apply equally to finite and infinite languages, we shall henceforth restrict our attention to machines having a finite number of internal locations, each taking on values from a finite domain. One immediate consequence is that all translation functions are computable, although complexity trade-offs remain. For instance, since all Boolean functions can be computed by a circuit of depth 2, we could always compute the translation function in constant time—if we were prepared to tolerate the potentially large number of computing elements that may be required. In the worst case, an exponential number of gates could be needed and the constant-time result is merely academic. Our aim, however, is to control the synthesis process in order to produce systems that not only track world conditions but are also practical to design and implement.

In the next section we discuss how informational data types can be used to approach the synthesis problem.

4 From analysis to synthesis: The machine induced by world dynamics

One way of using the situated-automata framework is for the analysis of existing machines: Given the description of an environment and of a machine embedded in that environment, we seek to describe the information encoded in its states. For purposes of design, however, we are more interested in the opposite question: Given a description of the world and of the information we would *like* to have encoded in machine states, how can we design the machine's circuitry in such a way that states of the machine will actually be correlated, as desired, with conditions in the world?

At the theoretical level, we can show that the dynamics of the world, together with the semantics of the machine's inputs and the *intended* semantics of its internal states (expressed as an informational data type), fully determine a machine whose

internal states carry the desired information by virtue of their actual correlation with the world. To see why this is the case, imagine we are given an informational data type $\tau_{in} = \langle D_{in}, M_{in}, R_{in} \rangle$ for the input location of a machine and an intended data type $\tau_a = \langle D_a, M_a, R_a \rangle$ for the internal location a . Imagine, further, that we are given the proposition φ_0 approximating the initial world condition (in the sense that φ_0 is implied by the true initial condition), and a function S approximating the true strongest-postcondition function (in the sense that $S(\varphi)$ is implied by the true strongest postcondition of φ for each φ ; approximation is the best we can do, since the world is not fully determined until we have fixed the embedded automaton.) We now show how these elements determine a machine that tracks changing world conditions as desired.

Here a machine will be defined by a pair of domains D_{in} and D_a (for inputs and internal states), an initial value $v_0 \in D_a$, and a next-state function $f : D_{in} \times D_a \rightarrow D_a$ satisfying the following conditions for all w, t :

$$\begin{aligned} w(a, 0) &= v_0 \\ w(a, t+1) &= f(w(in, t), w(a, t)) \end{aligned}$$

Given τ_{in} , τ_a , φ_0 , and S , we define v_0 and f as follows:

$$\begin{aligned} v_0 &= R_a(\varphi_0) \\ f(u, v) &= R_a(S(M_{in}(u) \cap M_a(v))) \end{aligned}$$

Intuitively, v_0 , the initial value of the location a , is just the representation, in a 's data type, of the initial proposition; the value of the next-state function is determined mathematically by considering the proposition associated with the old value of a and with the input, determining what will be true one time instant in the future given what is true now, and representing that proposition in the data type of a . Note the implicit reliance on spatial monotonicity and the similarity between this construction and the definition of translation functions in the previous section.

Assuming M_{in} is veridical, it can be shown that these definitions of v_0 and f insure that M_a will be veridical as well, i.e., that the machine's states will indeed be correlated with the intended meanings of those states. Mathematically, we must demonstrate that for all w, t :

$$M_a(w(a, t))(w, t).$$

The proof of this proposition is straightforward and proceeds by induction on t . (The variable w is universally quantified throughout.) The base case is established as follows: From the definition of φ_0 , we have that $\varphi_0(w, 0)$. The soundness of R_a gives us $M_a(R_a(\varphi_0))(w, 0)$, whence $M_a(w(a, 0))(w, 0)$ follows immediately from the definition of v_0 by simple substitution, since $w(a, 0) = v_0$.

The inductive case is established similarly. The induction hypothesis is that

$$M_a(w(a, t))(w, t),$$

and the veridicality of M_{in} gives us

$$M_{in}(w(in, t))(w, t).$$

Conjoining these conditions yields

$$(M_{in}(w(in, t)) \cap M_a(w(a, t)))(w, t).$$

The definition of S implies that

$$S(M_{in}(w(in, t)) \cap M_a(w(a, t)))(w, t + 1),$$

and the soundness of R_a guarantees that

$$M_a(R_a(S(M_{in}(w(in, t)) \cap M_a(w(a, t))))(w, t + 1).$$

Substituting in the definition of f , we get

$$M_a(f(w(in, t), w(a, t)))(w, t + 1),$$

from which $M_a(w(a, t + 1))(w, t + 1)$ follows immediately.

5 Syntactifying the construction

We have just seen how the dynamics of the environment, together with the semantics of the inputs and the intended semantics of the internal state, completely determine the structure of a machine. To be of practical utility, however, the mathematical construction must be made operational. One approach would be for the programmer, based on his intuitive understanding of the task environment, to define the induced automaton directly in a conventional programming language. Although adequate in principle, this approach is difficult to apply in practice for complex domains. For this reason we seek compilation techniques that would automate at least part of the synthesis process and make the transition from environment description to automaton more transparent to the programmer.

Although the automaton is mathematically determined by τ_{in} , τ_a , φ_0 , and S , we cannot directly present these abstract objects to a compiler and must use symbolic, often approximate, descriptions. Let us examine the form these descriptions might take for informational data types (τ_{in} and τ_a) and for world dynamics (φ_0 and S).

5.1 Specifying informational data types

Consider a machine location x of informational data type $\tau_x = \langle D_x, M_x, R_x \rangle$. Let us see how the three components of the data type might be described to a compiler.

The value domain D_x is straightforward to describe using conventional data type declarations. For our purposes, it will be sufficient to consider only atomic data types such as Booleans, integers, floats, etc. and record structures, possibly nested, over these atomic types. For example, to tell the compiler about the value domain of location x we might write x : `[bool [int int] float]`.

The specification of the other two components is more complex. Let us begin with M_x . Recall that M_x , the "meaning function" associated with location x , maps elements of D_x to Φ , the set of propositions, where propositions are modeled as sets of world-time pairs. Recall also that logical formulas can be used to partially express the content of a proposition, provided they are taken from a logic that assigns sets of world-time pairs as the denotation of formulas. Many temporal logics will suffice for this purpose. (For one example, see [Rosenschein and Kaelbling1986].) Given such a logic, formulas parametrized by run-time values can be used to define a meaning function from values to propositions.

To see this, let \mathcal{L} be the language of some temporal logic, with interpretation function $\mathcal{I} : \mathcal{L} \rightarrow \Phi$ and provability relation \vdash . Assume that among its individual terms, the language has constants that rigidly designate values of locations, possibly in addition to terms that denote location values that vary with world and time. If v is a data value, we let c_v stand for the rigid designator of value v in the language \mathcal{L} .

Let $P_x(U)$ be a formula of \mathcal{L} with a free variable U for which value-denoting terms can be substituted. Each substitution instance $P_x(c_v)$ is a closed formula to which the interpretation function \mathcal{I} can be applied. The parametric formula $P_x(U)$ thus induces a mapping $\hat{M}_x : D_x \rightarrow \Phi$ that approximates M_x and is defined as follows:

$$\hat{M}_x(v) = \mathcal{I}(P_x(c_v)).$$

The semantic interpretation \mathcal{I} of the language \mathcal{L} is itself approximated for the compiler by a set, Γ , of background facts relative to which the syntactic consequences of $P_x(c_v)$ are to be derived. To answer the question "does $M_x(u)$ imply $M_y(v)$," the compiler would attempt to establish $\Gamma \vdash P_x(c_u) \rightarrow P_y(c_v)$ using deductive techniques.

Having approximated M_x by a formula $P_x(U)$ and a background theory Γ , we have nearly determined the third component of the informational data type as well. Recall that the representation function R_x is intended to map propositions to elements of the value domain that best capture them. Since we are encoding propositions for the compiler using formulas, we are interested in functions that map formulas to data values. If Q is a formula expressing the proposition φ , candidate representations of φ (relative to Γ) should be drawn from the set

$$C_x(Q) = \{v \mid \Gamma \vdash Q \rightarrow P_x(c_v)\}.$$

Intuitively, elements of $C_x(Q)$ are the data values whose meaning is entailed by the information in Q , and hence by φ . This set must contain at least one element,

since there must be a value in D_x representing *true*, which is entailed by every proposition. In practice, it is convenient to have multiple representations of *true*, for instance by uniformly including a boolean *valid bit* in all data types. When the valid bit has the value zero, the meaning of the whole value is taken to be simply *true*, regardless of the values of the rest of the parameters.

If there is more than one element in $C_x(Q)$, the multiple values must somehow be combined so that they might "fit" into the space allotted to x . We call the functions responsible for combining these values *rconj* functions ("representation of the conjunction.") This function is defined as

$$rconj_x(v_1, v_2) = R_x(M_x(v_1) \cap M_x(v_2)).$$

Because the choice of R_x , and hence *rconj*, is under-determined by the meaning function M_x , the designer must somehow stipulate the *rconj* _{x} function directly for each type τ_x . This can be made relatively convenient through the use of declarative rules of the form

$$P_x(V_1) \wedge P_x(V_2) \rightarrow P_x(f(V_1, V_2)).$$

Having specified a binary *rconj* operator, arbitrary finite sets of values can be combined in the obvious way:

$$rconj_x^*(v_1, \dots, v_n) = \\ rconj_x(v_1, \dots, rconj_x(v_{n-1}, v_n) \dots)$$

The order of combination does not matter since the *rconj* function is commutative due to the commutativity of the underlying conjunction operator \cap in terms of which *rconj* is defined.

Example

We illustrate the concepts above by defining a sample informational data type. Consider a location x of value type [bool int]. Informally, the first field is the valid bit, and the second field is intended to represent a lower bound on the age of some individual, Fred.

The semantics of x can be expressed using the formula

$$P_x(U) = age(fred, [first(U), second(U)])$$

under the intended interpretation:

$$\mathcal{I}(age(fred, [V_1, V_2])) = \begin{cases} true & \text{if } V_1 = 0 \\ \varphi(V_2) & \text{if } V_1 = 1 \end{cases}$$

where $\varphi(V_2) = \{\langle w, t \rangle \mid age(fred, w, t) \geq \text{the number denoted by } V_2\}$. Thus, the run-time value $[0, n]$ at location x would represent the vacuous proposition *true* for any n , the value $[1, 14]$ would represent that Fred is at least 14 years old, and so on.

An *rconj* rule for the informational data type might look like this:

$$\begin{aligned} & age(fred, [U_1, U_2]) \wedge age(fred, [V_1, V_2]) \\ & \rightarrow age(fred, \\ & [or(U_1, V_1), if(U_1, if(V_1, max(U_2, V_2), U_2), V_2)]). \end{aligned}$$

This rule indeed defines a commutative *rconj* operator that can be used to combine values of this informational data type in a way consistent with the intended interpretation.

Furthermore, if the intended model incorporates the constraint that 18-year olds can vote, we might include among the background facts an assertion of the form

$$\begin{aligned} & age(fred, [U_1, U_2]) \rightarrow \\ & can-vote(fred, and(U_1, ge(U_2, 18))). \end{aligned}$$

This fact implicitly defines part of a translation function from the *age(fred, —)* data type to the *can-vote(fred, —)* data type. Notice that in this data type, *fred* is fixed at compile time. This would be appropriate if distinct locations were used to store information about individuals referred to explicitly at compile time. If Fred's identity were not known until run time, the run-time parameter would have to take on values that encoded propositions about Fred, Sam, etc.

5.2 Specifying World Dynamics

As we described above, the compiler is assumed to have access to a background theory, that is, a set of assertions describing the environment. This theory will contain many temporal facts as well as atemporal facts. By choosing the language \mathcal{L} to include appropriate temporal operators we can express facts about the initial condition of the world and about temporal transitions in a way that allows us to approximate the semantic objects φ_0 (initial condition) and S (strongest postcondition function.)

In the simplest case, the language \mathcal{L} need only include the modal operators \Box , *init*, and *next* satisfying the following semantic properties for all w, t :

$$w, t \models \Box \varphi \text{ iff } w', t' \models \varphi \text{ for all } w', t'$$

$$w, t \models \text{init } \varphi \text{ iff } w, 0 \models \varphi$$

$$w, t \models \text{next } \varphi \text{ iff } w, t + 1 \models \varphi$$

The compiler can answer questions of the form "does $S(M_x(u))$ imply $M_y(v)$ " by establishing $\Gamma \vdash \Box(P_x(c_u) \rightarrow \text{next } P_y(c_v))$ using deductive techniques.

5.3 Putting the Pieces Together: Synthesis Method, Abstract Version

We now describe a compilation method that operates on the representations discussed above and produces a circuit description of the desired automaton. The compiler takes as inputs a description of information carried by the run-time inputs to the machine and the internal machine state, as well as a background theory containing temporal facts. The compiler operates by deriving theorems about what is true initially and about what will be true next at any time, given what is true at that time. In the course of the derivation, free variables are instantiated in the manner of logic programming systems. From the instantiated formulas the compiler extracts the initial value of the machine's internal state and the description of a circuit for updating the machine's state vector.

More precisely, the compiler's inputs consist of the following:

- a list $[a_1, \dots, a_n]$ of input locations
- a list $[b_1, \dots, b_m]$ of internal locations
- for each input location a , a formula $P_a(U)$ with free variable U
- for each internal location b , a formula $P_b(U)$ with free variable U and a function $rconj_b$
- a finite set Γ of facts.

For each internal location b , the compiler computes two sets of value terms I_b and N_b defined as follows:

$$I_b = \{e \mid \Gamma \vdash \Box_{init} P_b(e)\}$$

$$N_b = \{e \mid \Gamma \vdash \Box_{P_{a_1}(a_1) \wedge \dots \wedge P_{b_n}(b_n)} \rightarrow next P_b(e)\}.$$

If these sets are infinite, they can be generated and used incrementally. This is discussed more fully below.

From these collections of sets the compiler computes the initial value and the update function. The initial value is computed as follows:

$$v_0 = [rconj_{b_1}^*(I_{b_1}), \dots, rconj_{b_m}^*(I_{b_m})],$$

In other words, the initial value of the state vector is the vector of values derived by *rconj*-ing values representing the strongest propositions that can be inferred by the compiler about the initial state of the environment in the "language" of each of the state components. Similarly, for the next-state function:

$$f([a_1, \dots, a_n], [b_1, \dots, b_m]) =$$

$$[rconj_{b_1}^*(N_{b_1}), \dots, rconj_{b_n}^*(N_{b_n})],$$

Here the compiler constructs a vector of expressions that denote the strongest propositions about what will be true next, again in the language of the state components.

In the case of the initial value, since all the terms are rigid, the *rconj* values can be computed at compile time. In the case of the next-state function, however, the *rconj* terms will not denote values known at compile time. Rather, they will generally be nested expressions containing operators that will be used to compute values at run time. Assuming the execution time of these operators is bounded, the depth of the expressions will provide a bound on the update time of the state vector.

Without restricting the background theory, we cannot guarantee that the sets I_b and N_b will be finite. However, even in the unrestricted case the finiteness of terms in the language guarantees that whichever elements we *can* derive at compile time can be computed in bounded time at run time. Furthermore, the synthesis procedure exhibits strongly monotonic behavior: the more elements of I_b and N_b we compute, the more information we can ascribe to run-time locations regarding the environment. This allows incremental improvements to be achieved simply by running the compiler longer; stopping the procedure at any stage will still yield a correct automaton, although not necessarily the automaton attuned to the most specific information available. Since, in general, additional *rconj* operations consume run-time resources, one reasonable approach would be to have the compiler keep track of run-time resources consumed and halt when some resource limit is reached.

As we have observed, without placing restrictions on the symbolic language used to specify the background theory, the synthesis method described above would hardly be practical; it is obvious that environment-description languages exist that make the synthesis problem not only intractable but undecidable. However, as with Gapps [Kaelbling1988] and other formalisms in the logic programming style, by restricting ourselves to certain stylized languages, practical synthesis techniques can be developed.

We have experimented with a restriction of the logical language that seems to offer a good compromise between expressiveness and tractability. This restriction is to a weak temporal Horn-clause language resembling Prolog but with the addition of *init* and *next* operators. In this language the background theory is given as facts of the following form:

```
init q(X,Y).
next q(X,f(X,Y)) :- q1(X,Y), ..., qn(X,Y).
q(X,f(X,Y)) :- q1(X,Y), ..., qk(X,Y).
```

For each predicate or function expression, the first argument represents a compile-time term and the second a run-time term. Facts of the first two sorts assert temporal facts (the \square operator is implicit), and facts of the last sort are ordinary

instantaneous facts, much as one would find in a conventional Prolog system, but with terms syntactically marked as compile-time or run-time.

The *rconj* rules are given in the following form:

$\text{rconj } q(x, f(x, y1, y2)) \text{ :- } q(x, y1), q(x, y2).$

The derivation process proceeds as described above but uses backward-chaining deduction techniques adapted from logic programming. Each distinct location has an associated atomic formula schema $p(i, Y)$. In deriving the initial value v_0 the compiler attempts to prove $p(i, Y)$ from the *init* declarations and the instantaneous facts. If this succeeds, the initial value of the i 'th component of the state vector is the *rconj** of the bindings of Y . If the attempt fails, the valid bit of that component is set initially to 0. Similarly, the next-state function for component i is derived by attempting to prove *next* $p(i, Y)$ using the *next* rules and the instantaneous facts, chaining backwards and cutting off proofs that traverse more than one "next" clause. Although the process of finding the proofs need not be real-time, the circuit that is finally produced is.

A prototype system, called RULER, has been built implementing the Horn-clause version of the synthesis algorithm. The language resembles Prolog in many ways, differing mainly in the strong distinction between compile-time and run-time expressions. Compile-time expressions undergo unification in the ordinary manner; run-time expressions, by contrast, are simply accumulated and used to generate the circuit description. RULER was implemented in Lisp as an extension of the Rex language [Kaelbling1987]. Run-time expressions in RULER are allowed to be any valid Rex expression, and all of the Rex optimizations (common-subexpression elimination, constant folding, etc.) are applied to the resulting circuit descriptions produced by RULER. The RULER system was run on several small examples involving object tracking and aggregation, and the synthesis procedure has proved tractable in our test implementation.

6 Future Directions

Our current research is directed toward extending the theoretical basis for synthesis and improving the practical utility of tools such as RULER. On the theoretical side, one important extension is to adapt the synthesis techniques to cases where the correlation between machine states and world conditions is best described probabilistically. Naive approaches will not work, primarily because the spatial-monotonicity property fails in the probabilistic case. For this reason we have been exploring design disciplines that reconcile structured synthesis methods with the inherently non-monotonic nature of probabilities, preserving the spirit of the techniques presented in this paper.

On the practical side, experiments with RULER suggest needed improvements in several areas. One syntactic improvement would be to uniformly suppress valid bits,

since their treatment is so systematic. Freer syntactic intermingling of compile-time and run-time expressions and tests would be useful as well. A more serious practical consideration has to do with helping the programmer control the combinatorics of the synthesis process. As in general logic programming, it is possible, using RULER, to write programs with unacceptable combinatorial behavior. While this is not the fault of RULER per se and can undoubtedly be ameliorated by increasing the programmer's experience and skill in using the tool, there are improvements that can be made in the system itself, including facilities for detecting cycles and redundant proofs. Finally, there is need to gain practical experience in applying this style of programming to real problems in visual perception, sensor fusion, and other similar areas.

7 Related Work

There has been considerable work on the synthesis of digital machines from temporal logic specifications, for example, the work by Ben Moszkowski [Moszkowski1983]. This work considers symbolic specifications similar to the kind considered here but does not connect them directly to an informational account of machine states. The work of Joseph Halpern and his associates [Halpern and Moses1985], on the other hand, has examined mathematical approaches closely related to our own for characterizing information in distributed system, but have so far not addressed issues of automated synthesis. Chris Goad [Goad1986] has used partial evaluation to generate efficient algorithms for visual recognition. Goad's techniques are rather domain-specific and do not handle tracking of conditions over time. There is a rich literature in the traditional AI paradigm (as well as in formal philosophy) on belief revision (see, for example [Doyle1979, de Kleer1986]), but little work has addressed the implications of real-time update requirements.

Acknowledgments

I have benefited greatly from discussions with Leslie Kaelbling, Michal Irani, and David Chapman.

References

- [Rosenschein1985] Rosenschein, Stanley J. "Formal Theories of Knowledge in AI and Robotics". In *New Generation Computing*, Vol. 3, No. 4, (special issue on Knowledge Representation), Ohmsha, Ltd., Tokyo, Japan (1985).

- [Rosenschein and Kaelbling1986] Rosenschein, Stanley J. and Leslie P. Kaelbling. "The Synthesis of Digital Machines with Provable Epistemic Properties," *Proceedings of Workshop on Theoretical Aspects of Reasoning About Knowledge*, Monterey, California (1986).
- [Kaelbling1988] Kaelbling, Leslie P. "Goals as Parallel Program Specifications." *Proceedings of the Seventh National Conference on Artificial Intelligence*, Morgan Kaufmann, St. Paul, Minnesota (August 1988).
- [Kaelbling1987] Kaelbling, Leslie P. "Rex: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems." *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts (1987).
- [Moszkowski1983] Moszkowski, Ben. Reasoning about Digital Circuits. Ph.D. Dissertation, Stanford University, Stanford, CA (1983).
- [Goad1986] Goad, Chris. "Fast 3D Model-Based Vision." In *From Pixels to Predicates: Recent Advances in Computational and Robotic Vision*, Alex P. Pentland (ed.), Ablex Publishing Corporation, Norwood New Jersey (1986).
- [Halpern and Moses1985] Halpern, Joseph Y. and Yoram Moses. "A guide to the modal logic of knowledge and belief." *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California (1985).
- [Doyle1979] Doyle, Jon. "A Truth Maintenance System." *Artificial Intelligence*, Vol. 12, No. 3 (1979).
- [de Kleer.1986] de Kleer, Johann. "An Assumption-Based Truth Maintenance System." *Artificial Intelligence*, Vol. 28, No. 1 (1986).

B Action and Planning in Embedded Agents

Action and Planning in Embedded Agents *

Leslie Pack Kaelbling
and
Stanley J. Rosenschein
Teleos Research

November 2, 1989

1 The Design of Embedded Agents

Embedded agents are computer systems that sense and act on their environments monitoring complex dynamic conditions and affecting the environment in goal-directed ways. Systems of this kind are extremely difficult to design and build, and without clear conceptual models and powerful programming tools, the complexities of the real world can quickly become overwhelming. In certain special cases, designs can be based on well-understood mathematical paradigms such as classical control theory. More typically, however, tractable models of this type are not available and alternative approaches must be used. One such alternative is the situated-automata framework, which models the relationship between embedded control systems and the external world in qualitative terms and provides a family of programming abstractions to aid the designer. This paper briefly reviews the situated-automata approach and then explores in greater detail one aspect of the approach, namely the design of the action-generating component of embedded agents.

1.1 The Situated-Automata Model

The theoretical foundations of the situated-automata approach are based on modeling the world as a pair of interacting automata, one corresponding to the physical environment and the other to the embedded agent. Each has local

*This work was supported in part by the Air Force Office of Scientific Research under contract F49620-89-C-0055DEF and in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC-2-494 through Stanford University subcontract PR-6359.

state that varies as a function of signals projected from the other. The aim of the design process is to synthesize an agent, in the form of an embedded state machine, that causes the desired effects in the environment over time.

In applications of interest, it is often useful to describe the agent in terms of the information available about the environment and the goals the agent is pursuing. It is also desirable that these descriptions be expressed in language that refers to states of the environment rather than to specific internal data structures, at least during the early phases of design. Moreover, the inputs, outputs, and internal states of the state machine will be far too numerous to consider explicitly, which means the machine must be constructed out of a set of separate components acting together to generate complex patterns of behavior. These requirements highlight the need for compositional, high-level languages that compactly describe machine components in semantically meaningful terms.

Situated-automata theory provides a principled way of interpreting data values in the agent as encoding facts about the world expressed in some language whose semantics is clear to the designer. Interpretations of this sort would be of little use were it not also the case that whenever the data structure had a particular value, the condition denoted was guaranteed to hold in the environment. Such considerations motivate defining the semantics of data structures in terms of objective correlations with external reality. In this approach, a machine variable x is said to carry the information that p in world state s , written $s \models K(x, p)$, if for all world states in which x has the same value it does in s , the proposition p is true. The formal properties of this model and its usefulness for programming embedded systems have been described elsewhere [9,11,5,10].

Having established a theoretical basis for viewing a given signal or state in the agent as carrying information content by virtue of its objective correlation the environment, one can consider languages in which this content might be expressed. In general there will be no single "best" language for expressing this information. For example, one language is the set of signals or states themselves. These can be regarded as a system of signs whose semantic interpretations are exactly the conditions with which they are correlated. However, the designer will typically wish to employ other, higher-level, languages during the design process. This theme will be expanded upon below in connection with goal-description languages.

1.2 Perception-Action Split

One way of structuring the design process for the cognitive ease of the designer is to separate the problem of acquiring information about the world from the problem of acting appropriately relative to that information. The former we shall label *perception* and the latter, *action*. In terms of the state-machine model, as shown in Figure 1, the perception component corresponds to the update function and the initial state, whereas the action component corresponds to the output mapping.

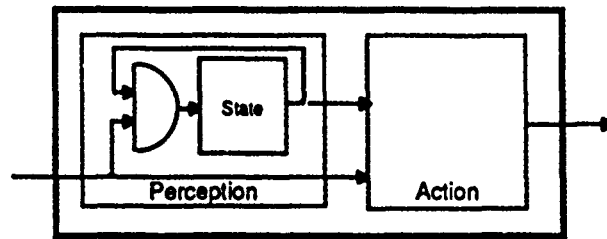


Figure 1: Division between perception and action components.

The perception-action split in itself is entirely conceptual and may or may not be the basis for modularizing the actual system. Horizontal decompositions that cut across perception and action have been advocated by Brooks as a practical way of approaching agent design [2]. The horizontal approach allows the designer to consider simultaneously those limited aspects of perception and action needed to support specific behaviors. In this way, it discourages the pursuit of spurious generality that often inhibits practical progress in robotics.

These attractive features are counterbalanced, however, by the degree to which horizontal decomposition encourages linear thinking. In practice, the methodology of not separating the acquisition of information from its use tends to encourage the development of very specific behaviors rather than the identification of elements that can recombine freely to produce complex *patterns* of behavior. The alternative is a *vertical* strategy based on having separate system modules that recover broadly useful information from multiple sources and others that exploit it for multiple purposes. The inherent combinatorics of information extraction and behavior generation make the vertical approach attractive as a way of making efficient use of a programmer's effort.

The commitment to a decomposition based upon the perception-action split still leaves open the question of development strategy. One approach is to iteratively refine the perception-action pair, more or less in lockstep. The information objectively carried by an input signal or an internal state is relative to constraints on other parts of the system—including constraints on the action component. The more constrained the rest of the system, the more the designer can deduce about the world from a given internal signal or state, hence the more "information" it contains. As the designer refines his design, his model of the information available to the system and what the system will do in response becomes increasingly specific.

An alternative to iterative refinement, suitable in many practical design situations, is the strict divide-and-conquer strategy in which the design of the perception component is carried out in complete isolation from the development of the action component except for the specification of a common interface—the data structures that encode the information shared between the perception and action modules. Although there may be occasions when the designer needs to

rely on some fact about what the agent will *do* in order to guarantee that a certain signal or state has the semantic content he intends, if these situations can be minimized or ignored, considerable simplification will result.

1.3 Goals

As we have seen, one way of semantically characterizing an agent's states is in terms of the information they embody. The perception component delivers information, and the action component maps this information to action. In many cases, however, it is more natural to describe actions as functions not only of information but of the goals the agent is pursuing at the moment [12].

Goals can be divided into two broad classes: static and dynamic. A static goal is a statement the agent's behavior is simply designed to make true. In reality, a static goal is nothing more than a specification, and as such the attribution of this "goal" to the agent is somewhat superfluous, although it may be of pragmatic use in helping the designer organize his conception of the agent's action strategy. Dynamic goals are another matter. The ability to attribute to the agent goals that change dynamically at run time opens the possibility of dramatically simplifying the designer's description of the agent's behavior.

Since we are committed to an information-based semantics for reactive systems, we seek an "objective" semantics of goals defined explicitly in informational terms. We can reformulate the notion of having a goal p as having the information that p implies a fixed top-level goal, called N for "Nirvana." Formally, we define a goal operator G as follows:

$$G(x, p) \equiv K(x, p \rightarrow N) .$$

In this model, x has the goal p if x carries the information that p implies Nirvana.¹ This definition captures the notion of dynamic goals because p can be an indexical statement, such as "it is raining now," whose truth varies with time. Since this model defines goals explicitly in terms of information, the same formal tools used to study information can be applied to goals as well. In fact, under this definition, goals and information are dual concepts.

To see the duality of goals and information, consider a function f mapping values of one variable, a , to values of another variable, b . Under the information interpretation, such a function takes elements having more specific information into elements having less specific information. This is because functions generally introduce ambiguity by mapping distinct inputs to the same output. For example, if value u_1 at a is correlated with proposition p and value u_2 at a is correlated with q and if f maps both u_1 and u_2 to v at b , the value v is ambiguous as to whether it arose from u_1 or u_2 , and hence the information it contains

¹We observe that under this definition *False* will always be a goal; in practice, however, we are only interested in non-trivial goals.

is the disjunctive information $p \vee q$, which is less specific than the information contained in either u_1 or u_2 . Thus, functional mappings are a form of forgetting.

Under the goal interpretation, this picture is reversed. The analog to "forgetting" is committing to subgoals, which can be thought of as "forgetting" that there are other ways of achieving the condition. For instance, let the objective information at variable a be that the agent is hungry and that there is a sandwich in the right drawer and an apple in the left. If the application of a many-to-one function results in variable b 's having a value compatible with the agent's being hungry and there being a sandwich in the right drawer and either an apple in the left drawer or not, we could describe this state of affairs by saying that variable b has lost the information that opening the left drawer would be a way of finding food. Alternatively, we could say that variable b had committed to the subgoal of opening the right drawer. The phenomena of forgetting and commitment are two sides of the same coin.

We can relate this observation to axioms describing information and goals. One of the formal properties satisfied by K is the deductive closure axiom, which can be written as follows:

$$K(x, p \rightarrow q) \rightarrow (K(x, p) \rightarrow K(x, q)) .$$

The analogous axiom for goals is

$$K(x, p \rightarrow q) \rightarrow (G(x, q) \rightarrow G(x, p)) .$$

This is precisely the subgoaling axiom. If the agent has q as a goal and carries the information that q is implied by some other, more specific, condition, p , the agent is justified in adopting p as a goal. The validity of this axiom can be established directly from the definition of G .

Given these two ways of viewing the semantics of data structures, we can revisit the state-machine model of agents introduced above. Rather than specify the action component of the machine as a function of one argument interpreted in purely "informational" terms, $f(i)$, it may be much more convenient for designers to define it as a function of two arguments, $f'(g, i)$ where the g argument is interpreted as representing the dynamic goals of the agent. Where does the g input come from? Clearly, it must ultimately be computed from the agent's current information state as well as its static goals, g_0 . As such, it must be equivalent to some non-goal-dependent specification: $f(i) = f'(\text{extract}(i, g_0), i)$. Nevertheless, the decomposition into a goal-extraction module and a goal-directed action module may significantly ease the cognitive burden for the designer while leaving him secure in the knowledge that it is semantically grounded.

1.4 Software Tools for Agent Design

Although it is conceptually important to have a formal understanding of the semantics of the data structures in an embedded agent, this understanding does

not, directly, simplify the programmer's task. For this reason, it is necessary to design and implement software tools that are based on proper foundations and that make it easier to program embedded agents.

Rex [5,7] is a language that allows the programmer to use the full recursive power of Lisp at compile time to specify a synchronous digital circuit. The circuit model of computation facilitates semantic analysis in the situated-automata theory framework. Rex provides, however, a low-level, operational language that is more akin to standard programming languages than to declarative AI languages. For this reason, we have designed and implemented a pair of declarative programming languages on top of the base provided by Rex. Ruler [10] is based on the "informational" semantics and is intended to be used to specify the perception component of an agent. Gapps [6] is based on the "goal" semantics and is intended to be used to specify the action component of an agent. In the rest of this paper, we will describe the Gapps language, its use in programming embedded agents, and a number of extensions that relate it to more traditional work in planning.

2 Gapps

In this section we describe Gapps, a language for specifying behaviors of computer agents that retains the advantage of declarative specification, but generates run-time programs that are reactive, do parallel actions, and carry out strategies made up of very low-level actions.

Gapps is intended to be used to specify the action component of an agent. The Gapps compiler takes as input a declarative specification of the agent's top-level goal and a set of goal-reduction rules, and transforms them into the description of a circuit that has the output of the perception component as its input, and the output of the agent as a whole as its output. The output of the agent may be divided into a number of separately controllable actions, so that we can independently specify procedures that allow an agent to move and talk at the same time. A sample action vector declaration is:

```
(declare-action-vector
  (left-wheel-velocity int)
  (right-wheel-velocity int)
  (speech string))
```

This states that the agent has three independently controllable effectors and declares the types of the output values that control them.

In the following sections, we shall present a formal description of Gapps and its goal evaluation algorithm, and explain how Gapps specifications can be instantiated as circuit descriptions.

2.1 Goals and Programs

The Gapps compiler maps a top-level goal and a set of goal-reduction rules into a program. In this section we shall clarify the concepts of goal, goal-reduction rule, and program.

There are three primitive goal types: goals of execution, achievement, and maintenance. Goals of execution are of the form $\text{do}(a)$, with a specifying an instantaneous action that can be taken by the agent in the world—the agent's goal is simply to perform that action. If an agent has a goal of maintenance, notated $\text{maint}(p)$, then if the proposition p is true, the agent should strive to maintain the truth of p for as long as it can. The goal $\text{ach}(p)$ is a goal of achievement, for which the agent should try to bring about the truth of proposition p as soon as possible. The set of goals is made up of the primitive goal types, closed under the Boolean operators. The notions of achievement and maintenance are dual, so we have $\neg\text{ach}(p) \equiv \text{maint}(\neg p)$ and $\neg\text{maint}(p) \equiv \text{ach}(\neg p)$.

In order to characterize the correctness of programs with respect to the goals that specify them, we must have a notion of an action *leading to* a goal. Informally, an action a leads to a goal G (notated $a \leadsto G$) if it constitutes a correct step toward the satisfaction of a goal. For a goal of achievement, the action must be consistent with the goal condition's eventually being true; for a goal of maintenance, if the condition is already true, the action must imply that it will be true at the next instant of time. The *leads to* operator must also have the following formal properties:

$$\begin{aligned} a &\leadsto \text{do}(a) \\ (a \leadsto G) \wedge (a \leadsto G') &\Rightarrow a \leadsto (G \wedge G') \\ (a \leadsto G) \vee (a \leadsto G') &\Rightarrow a \leadsto (G \vee G') \\ \text{cond}(p, a \leadsto G, a \leadsto G') &\Rightarrow a \leadsto \text{cond}(p, G, G') \\ (a \leadsto G) \wedge (G \rightarrow G') &\Rightarrow a \leadsto G' \end{aligned}$$

This definition captures a weak intuition of what it means for an action to lead to a goal. The goal of doing an action is immediately satisfied by doing that action. If an action leads to each of two goals, it leads to their conjunction; similarly for disjunction and conditionals. The definition of *leads to* for goals of achievement may seem too weak—rather than saying that doing an action is consistent with achieving the goal, we would like somehow to say that the action actually constitutes *progress* toward the goal condition. Unfortunately, it is difficult to formalize this notion in a domain-independent way. In fact, any definition of *leads to* that satisfies this definition is compatible with the goal reduction algorithm used by Gapps, so the definition may be strengthened for a particular domain.

Goal reduction rules are of the form $(\text{defgoal } G \ G')$ and have the semantics that the goal G can be reduced to the goal G' ; that is, that G' is a

specialization of G , and therefore implies G . In this case, any action that leads to G' will also lead to G .

A program is a finite set of condition-action pairs, in which the condition is a run-time expression (actually a piece of Rex circuitry with a Boolean-valued output) and an action is a vector of run-time expressions, one corresponding to each primitive output field. These actions are run-time mappings from the perceptual inputs into output values, and can be viewed as strategies, in which the particular output to be generated depends on the external state of the world via the internal state of the agent. Allowing the actions to be entire strategies is very flexible, but makes it impossible to enumerate the possible values of an output field. In order to specify a program that controls only the speech field of an action vector, we need to be able to describe a program that requires the speech field to have a certain value, but makes no constraints on the values of the other fields. One way to do this would be to enumerate a set of action vectors with the specified speech value, each of which has different values for the other action vector components. Instead of doing this, we allow elements of an action vector to contain the value \emptyset , which stands for all possible instantiations of that field.

A program Π , consisting of the condition-action pairs $\{\langle c_1, a_1 \rangle, \dots, \langle c_n, a_n \rangle\}$, is said to *weakly satisfy* a goal G if, for every condition c_i , if that condition is true, the corresponding action a_i leads to G . That is,

$$\Pi \text{ weakly satisfies } G \iff \forall i. c_i \rightarrow (a_i \leadsto G).$$

Note that the conditions in a program need not be exhaustive—satisfaction does not require that there be an action that leads to the goal in every situation, since this is impossible in general. We will refer to the class of situations in which a program does specify an action as the *domain* of the program. We define the domain of Π as

$$\text{dom}(\Pi) = \bigvee_i c_i.$$

A goal G is *strongly satisfied* by program Π if it is weakly satisfied by Π and $\text{dom}(\Pi) = \text{true}$; that is, if for every situation, Π supplies an action that leads to G . The conditions in a program need not be mutually exclusive. When more than one condition of a program is true, the action associated with each of them leads to the goal, and an execution of the program may choose among these actions nondeterministically.

Given the non-deterministic execution model, we can give programs a declarative semantics, as well. A program $\Pi = \{\langle c_1, a_1 \rangle, \dots, \langle c_n, a_n \rangle\}$, can be thought of as having the logical interpretation

$$\left(\bigwedge_i (a_i \rightarrow c_i) \wedge \bigvee_i a_i \right) \vee \neg \bigvee_i c_i.$$

Either the domain of the program is false (the second clause) or there is some action that is executed and the condition associated with that action is true.

2.2 Recursive Goal Evaluation Procedure

Gapps is implemented on top of Rex, and makes use of constructs from the Rex language to provide perceptual tests. There is not room here to describe the details of the Rex language, so we refer the interested reader to other papers [5,7]. Gapps programs are made up of a set of goal reduction rules and a top-level goal-expression. The general form of a goal-reduction rule is

(defgoalr *goal-pat goal-expr*),

where

```
goal-pat ::= (ach pat rex-params )
           (maint pat rex-params )

goal-expr ::= (do index rex-expr )
              (and goal-expr goal-expr )
              (or goal-expr goal-expr )
              (not goal-expr )
              (if rex-expr goal-expr goal-expr )
              (ach pat rex-expr )
              (maint pat rex-expr )
```

index is a keyword, *pat* is a compile-time pattern with unifiable variables, *rex-expr* is a Rex expression specifying a run-time function of input variables, and *rex-params* is a structure of variables that becomes bound to the result of a *rex-expr*. The details of these constructs will be discussed in the following sections.

The Gapps compiler is an implementation of an evaluation function that maps goal expressions into programs, using a set of goal reduction rules supplied by the programmer. In this section we shall present the evaluation procedure; we have shown that it is correct; that is, that given a goal G and a set of reduction rules Γ , $\text{eval}(G, \Gamma)$ weakly satisfies G .

Given a reduction-rule set Γ , we define the evaluation procedure as follows:

```
define eval(G)
case first(G)
do : make-primitive-program(second(G),third(G))
and: conjoin-programs(eval(second(G)),eval(third(G)))
or : disjoin-programs(eval(second(G)),eval(third(G)))
not: eval (negate-goal-expr(second(G)))
if : disjoin-programs
      (conjoin-cond(second(G),eval(third(G))),
       conjoin-cond(negate-cond(G),eval(fourth(G))))
```

```

maint,
ach: for all R in Gamma such that match(G, head (R))
      disjoin-programs(eval(body(R))

```

We shall now consider each of these cases in turn.

Do

The function **make-primitive-program** takes an index and a Rex expression and returns a program. The index indicates which of the fields of the action vector is being assigned, and the Rex expression denotes a function from the input to values for that action field. It is formally defined as

$$\text{make-primitive-program}(i, \text{rex-expr}) = \{(true, (\emptyset, \dots, \text{rex-expr}, \dots, \emptyset))\},$$

with the *rex-expr* in the *i*th component of the action vector. This program allows any action so long as component *i* of the action is the strategy described by *rex-expr*.

And

Programs are conjoined by taking the cross-product of their condition-action pairs and merging each of elements of the cross-product together. In conjoining two programs, the merged action vector is associated with the conjunction of the conditions of the original pairs, together with the condition that the two actions are mergeable. The conjunction procedure simply finds the pairs in each program that share an action and conjoins their conditions. We can define the operation formally as

$$\text{conjoin-programs}(\Pi', \Pi'') = \{(\langle c'_i \wedge c''_j \wedge \text{mergeable}(a'_i, a''_j) \rangle, \text{merge}(a'_i, a''_j))\}$$

for $1 \leq i \leq m, 1 \leq j \leq n$ where

$$\begin{aligned} \Pi' &= \{\langle c'_1, a'_1 \rangle, \dots, \langle c'_m, a'_m \rangle\} \\ \Pi'' &= \{\langle c''_1, a''_1 \rangle, \dots, \langle c''_n, a''_n \rangle\}. \end{aligned}$$

The conjunction operation preserves the declarative semantics of programs; that is, the semantic interpretation of the conjoined program is implied by the conjunction of the semantic interpretations of the individual programs.

Two action vectors are *mergeable* if, for each component, at least one of them is unspecified or they are equal.

$$\begin{aligned} \text{mergeable}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle) \equiv \\ \forall i. (a_i = \emptyset) \vee (b_i = \emptyset) \vee (a_i = b_i). \end{aligned}$$

If either component is unspecified, the test can be completed at compile time and no additional circuitry is generated. Otherwise, an equality test is conjoined in with the conditions to be tested at run time.

Action vectors are merged at the component level, taking the defined element if one is available. If the vectors are unequally defined on a component, the result is undefined:

$\text{merge}(\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle) = \langle c_1, \dots, c_n \rangle$, where

$$c_i = \begin{cases} a_i & \text{if } b_i = \emptyset \text{ or } b_i = a_i \\ b_i & \text{if } a_i = \emptyset \\ \perp & \text{otherwise.} \end{cases}$$

The merger of two action vectors results in an action vector that allows the intersection of the actions allowed by the original ones.

Or

The disjunction of two programs is simply the union of their sets of condition-action pairs. Stated formally,

$$\text{disjoin-programs } (\Pi', \Pi'') = \Pi' \cup \Pi''.$$

Not

In Gapps, negation is driven into an expression as far as possible, using De-Morgan's laws and the duality of *ach* and *maint*, until the only expressions containing *not* are those of the form *(ach (not pat))*, *(maint (not pat))*, and *(not (do index rex-expr))*. In the first two cases, there must be explicit reduction rules for the goal; in the last case we simply return the empty program. The handling of negation could be much stronger if we provided for the enumeration of all possible values of any action vector component and required them to be known constants at compile time. Then *(not (do left-velocity 6))* would be the same as $\bigvee_{i \neq 6} \text{make-primitive-program}(\text{left-velocity}, i)$; that is, license to go at any velocity but 6. As we noted before, these limitations are too severe for use in controlling a complex agent that has large numbers of possible outputs.

The procedure *negate-goal-expression* rewrites goal expressions as follows:

$$\begin{aligned} (\text{not } (\text{and } G_1 \ G_2)) &\Rightarrow (\text{or } (\text{not } G_1) \ (\text{not } G_2)) \\ (\text{not } (\text{or } G_1 \ G_2)) &\Rightarrow (\text{and } (\text{not } G_1) \ (\text{not } G_2)) \\ (\text{not } (\text{not } G)) &\Rightarrow G \\ (\text{not } (\text{if } c \ G_1 \ G_2)) &\Rightarrow (\text{if } c \ (\text{not } G_1) \ (\text{not } G_2)) \\ (\text{not } (\text{ach } p)) &\Rightarrow (\text{maint } (\text{not } p)) \\ (\text{not } (\text{maint } p)) &\Rightarrow (\text{ach } (\text{not } p)) \end{aligned}$$

If

The evaluation procedure for conditional programs hinges on the definition of the conditional operator $\text{cond}(p, q, r)$ as $(p \wedge q) \vee (\neg p \wedge r)$. The procedure for conjoining a condition and a program is defined as follows:

$$\text{conjoin-cond}(p, \Pi) = \{(p \wedge c_1, a_1), \dots, (p \wedge c_n, a_n)\}.$$

Thus,

$$\text{disjoin-programs}(\text{conjoin-cond}(p, \Pi'), \text{conjoin-cond}(\neg p, \Pi'')) = \{(p \wedge c'_1, a'_1), \dots, (p \wedge c'_n, a'_n), (\neg p \wedge c''_1, a''_1), \dots, (\neg p \wedge c''_m, a''_m)\}.$$

Ach and Maint

Goals of maintenance and achievement are evaluated by disjoining the results of al. applicable reduction rules in the rulebase Γ . A reduction rule whose head is the expression $(\text{ach } pat_1 \text{ rez-params})$ matches the goal expression $(\text{ach } pat_2 \text{ rez-expr})$ if pat_1 and pat_2 can be unified in the current binding environment. The patterns are s-expressions with compile-time variables that are marked by a leading ?. The Rex expression and parameter arguments may be omitted if they are null. The binding environment consists of other bindings of compile-time variables within the goal expression being evaluated. Thus, when evaluating the $(\text{ach } (\text{go } ?p))$ subgoal of the goal $(\text{and } (\text{ach } (\text{drive } ?q ?p)) (\text{ach } (\text{go } ?p)))$, we may already have a binding for ?p. As in Prolog, evaluation of this goal will backtrack through all possible bindings of ?p and ?q.

Once a pattern has been matched, Gapps sets up a new compile-time binding environment for evaluating the body of the rule. This is necessary in case variables in the body are bound by the invocation, as in

```
(defgoalr (ach (at ?p) [dist-err angle-err])
  (if (not-facing ?p angle-err)
    (ach (facing ?p) angle-err)
    (ach (moved-toward ?p) dist-err))) .
```

In the rule above, $(\text{at } ?p)$ is a pattern, ?p is a compile-time parameter, dist-err and angle-err are Rex variables, and $(\text{not-facing } ?p \text{ angle-err})$ will be a Rex expression once a binding is substituted for ?p. A possible invocation of this rule would be:

```
(ach (at (office-of stan)) [*distance-eps* 10]) .
```

Gapps also creates a new Rex-variable binding environment when the rule is invoked, binding the Rex variables in the head to the evaluated Rex expressions in the invocation. These variables may appear in Rex expressions in the body of the rule. Note that compile-time variables may also be used in Rex expressions, in order to choose at compile time from among a class of available run-time functions.

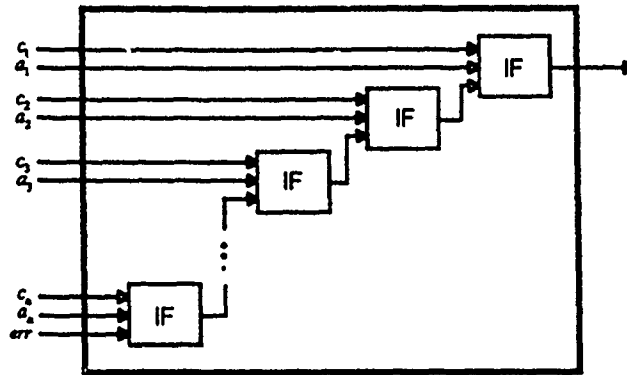


Figure 2: Circuit generated from Gapps program

2.3 Generating a Circuit

Once a goal expression has been evaluated, yielding a program, a circuit similar to the one shown in Figure 2, that instantiates the program is generated.² Because any action whose associated condition is true is sufficient for correctness, the conditions are tested in an arbitrary order that is chosen at compile time. The output of the circuit is the action corresponding to the first condition that is true. If no condition is satisfied, an error action is output to signal the programmer that he has made an error. If, at the final stage of circuit generation, there are still \emptyset components in an action vector, they must be instantiated with an arbitrary value. The inputs to the circuit are computed by the Rex expressions supplied in the *if* and *do* forms. The outputs of the circuit are used to control the agent.

2.4 Reducing Conjunctive Goal Expressions

Conjunctive goal expressions can have two forms: (*ach-or-maint* (*and* p_1 p_2)) and (*and* (*ach-or-maint* p_1) (*ach-or-maint* p_2)). Because of the properties of maintainance, the goals (*maint* (*and* p_1 p_2)) and (*and* (*maint* p_1) (*maint* p_2)) are semantically equivalent. This is not true, however, for goals of achievement. The goal (*ach* (*and* p_1 p_2)) requires that p_1 and p_2 be true simultaneously, whereas the goal (*and* (*ach* G_1) (*ach* G_2)) requires only that they each be true at some time in the future.

Goals of the form (*ach-or-maint* (*and* p_1 p_2)) can only be reduced using reduction rules whose pattern matches this conjunctive pattern. Goals of the form (*and* (*ach-or-maint* p_1) (*ach-or-maint* p_2)) can be reduced in two

²An equivalent, but more confusing, circuit with $\log(n)$ depth can be generated for improved performance on parallel machines.

ways: using the standard evaluation procedure for conjunctive goals and using special reduction rules. It is often the case that an effective behavior for achieving G_1 and achieving G_2 cannot be generated simply by conjoining programs that achieve G_1 and G_2 individually. A program for the goal (and (ach have hammer) (ach have saw)) will almost certainly be incomplete when the two tools are in different rooms, because there will be no actions available that are consistent with the standard programs for achieving each of the subgoals. Because of this, we allow reduction rules of the form (defgoalr (and (ach-or-maint pat_1 rez-params $_1$) (ach-or-maint pat_2 rez-params $_2$)) goal-expr) so that special behaviors can be generated in the face of a conjunctive goal.

Following is an example that illustrates both kinds of conjunctive goals. At the top level, the goal is to have the hammer and saw simultaneously, but this reduces to conjunctions of ach and maint goals.

```
(defgoalr (ach (and (have hammer) (have saw))
  (if (have hammer)
    (and (maint have hammer) (ach have saw))
    (if (have saw)
      (and (maint have saw) (ach have hammer))
      (if (closer-than hammer saw)
        (ach have hammer)
        (ach have saw))))))
```

The agent will pursue the closer object until he has it, then pursue the second while maintaining possession of the first. We might need a similar rule for reducing the conjunctions of goals of achievement and maintenance. Instead of the specific rule above, we could write a more generic sequencing rule, like the following:

```
(defgoalr (ach (and ?g1 ?g2) [g1-params g2-params])
  (if (holds ?g1 g1-params)
    (and (maint ?g1 g1-params) (ach ?g2 g2-params))
    (if (holds ?g2 g2-params)
      (and (maint ?g2 g2-params)
        (ach ?g1 g1-params))
      (if (better-to-pursue ?g1 g1-params
        ?g2 g2-params)
        (ach ?g1 g1-params)
        (ach ?g2 g2-params)))))) .
```

The generic form of the rule assumes that there is a Rex function, holds, that takes a compile-time parameter and generates a circuit that tests to see whether the predicate encoded by the compile-time parameter and the run-time variables is true in the world.

2.5 Prioritized Goal Lists

It is often convenient to be able to specify a prioritized list of goals. In Gapps, we can do this with a goal expression of the form (prio goal-expr $_1$... goal-expr $_n$).

The semantics of this is

$$\text{cond}(\text{dom}(\Pi_1), \Pi_1, \\ \text{cond}(\text{dom}(\Pi_2), \Pi_2, \dots, \\ \text{cond}(\text{dom}(\Pi_{n-1}), \Pi_{n-1}, \Pi_n) \dots)),$$

where $\Pi_i = \text{eval}(\text{goal-expr}_i)$. The domain of a program (true in a situation if the program has an applicable action in that situation) is the disjunction of the conditions in the program. A program for a *prio* goal executes the first program, unless it has no applicable action, in which case it executes the second program, and so on. At circuit-generation time, this construct can be implemented simply by concatenating the programs in priority order, and executing the first action whose corresponding condition is satisfied.

An example of the use of the *prio* construct comes about when there is more than one way of achieving a particular goal and one is preferable to the other for some reason, but is not always applicable. We might have the rule

```
(defgoalr (ach in-room r)
  (prio (ach follow-planned-route-to r)
        (ach use-local-navigation-to r))) .
```

This rule states that the agent should travel to rooms by following planned paths, but if for some reason it is impossible to do that, it should do so through local navigation. The same effect could be achieved with an *if* expression, but this rule does not require the higher-level construct to know the exact conditions under which the higher-priority goal will fail.

2.6 Prioritized Conjunctions

An interesting special case of a prioritized set of goals is a prioritized conjunction of goals, in which the most preferred goal is the entire conjunction, and the less preferred goals are the conjunctions of shorter and shorter prefixes of the goal sequence. We define (*prio-and* $G_1 G_2 \dots G_n$) to be

```
(prio (and G1 G2 ... Gn)
      (and G1 G2 ... Gn-1) ...
      (and G1 G2)
      G1).
```

Isaac Asimov's three laws of robotics [1] are a well-known example of this type of goal structure. As another example, consider a robot that can talk and push blocks. It has as its top-level goal

```
(prio-and (maint not-crashed)
  (ach (in block1 room3))
  (maint humans-not-bothered)) .
```

It also has rules that say that any action with the null string in the talking field will maintain **humans-not-bothered**; that (in $?x ?y$) can be achieved by pushing $?x$ or by asking a human to pick it up and move it; and that any action that keeps the robot from coming into contact with a wall will maintain **not-crashed**. As long as the robot can push the block, it can satisfy all three conditions. If, however, the block is in a corner, getting in a position to push it would require sharing space with a wall, thus violating the first subgoal. The most preferred goal cannot be achieved, so we consider the next-most-preferred goal, obtained by dropping the last condition from the conjunction. Since it is now allowed to bother humans, the robot can satisfy its goal by asking someone to move the block for it. As soon as the human complies, moving the block out of the corner, the robot will automatically revert to its former pushing behavior. This is a convenient high-level construct for programming flexible reactive behavior without the need for the programmer to explicitly envision every combination of conditions in the world. It is important to remember that all of the symbolic manipulation of the goals happens at compile time; at run time, the agent simply executes the action associated with the first condition that evaluates to true.

3 Extending Gapps

Gapps is an appropriate language for specifying action maps that can be hard-wired at the compile time of the agent. In this section, we will consider ways of extending and augmenting Gapps to do exhaustive planning at compile time, to do run-time planning, and to do run-time goal reduction.

3.1 Universal Planning with Goal-Reduction Schema

Schoppers [13] has introduced the notion of a *universal plan*. A universal plan is a function that, for a given goal, maps every possible input situation of the agent into an action that leads to (in an informal sense) that goal. The program resulting from the Gapps-evaluation of a goal can be thought of as a universal plan, mapping situations to actions in service of the top-level goal.

Schoppers' approach differs from Gapps in that the user specifies the capabilities of the agent in an operator-description language. This language allows the user to specify a set of atomic capabilities of the agent, called operators, and the expected effect that executing each of the operators will have on the world, depending possibly on the state of the world in which the operator was executed.

Another way to characterize operators is through the use of a *regression function* [8]. The relation $q = \text{regress}(\alpha, p)$ holds if, whenever q holds in the world, the agent's performing action α will cause p to hold in the world as a result. In general, the regression function will return the weakest such q . Regres-

sion is usually used to look backwards from a goal-situation p ; the proposition q describes a set of situations that are only one "step" or operator application away from the set of situations satisfying p . We know that if the agent can get to a situation satisfying q , it can easily get to a situation satisfying p .

The following schematic Gapps rule allows it to do the exhaustive backward-chaining search that is typically done by a planner, in order to construct a universal plan. The Gapps compiler must be augmented slightly by giving it a depth-bound for its backward chaining, because this rule would, by default, cause infinite backward chaining.

```
(defgoalr (ach (before ?p ?q))
  (if (holds ?q)
    fail
    (if (holds ?p)
      (do anything)
      (if (holds (regress ?a ?p))
        (do ?a)
        (ach (before (regress ?a ?p) (regress ?a ?q)))))))
```

The reduction rule is for goals of the form $(ach (before ?p ?q))$; that is, the goal is to achieve some condition $?p$ before some other condition $?q$ obtains. This form of achievement goal is, we think, typical—it is rare that an agent has a goal of achieving something no matter how long it takes. The rule works as follows: if $?q$ is true in the world, the agent fails; if $?p$ holds in the world, then the agent can do anything because it has achieved its goal; otherwise, if, for any action $?a$, $(regress ?a ?p)$ holds (that is, performing action $?a$ will cause $?p$ to hold next time) then this goal reduces to the goal $(do ?a)$; finally, this goal can be reduced to achieving, for any action $?a$, $(before (regress ?a ?p) (regress ?a ?q))$. The final reduction says that it is good for the agent to get into a state from which action $?a$ achieves the goal $?p$ before the agent gets into a state from which action $?a$ achieves the releasing condition $?q$, because once that has been done, all the agent must do is do action $?a$.

Consider the application of this process to the standard 3-block blocks-world problem. The actions are named atoms, like pab , which signifies "put a on b." The world is described by predicates like ca , which signifies "clear a" and obt , which signifies "on b table." An additional predicate, $time(i)$, is true if the time on some global clock, which starts at 0, is i . We will use the abbreviation t_i to stand for $time(i)$. Given the goal $(ach (before (and oab obc) (time 2)))$, the evaluation procedure returns a program that is described propositionally as follows:

```
{{(¬t2 ∧ obc ∧ ca ∧ cb), pab},
 {{(¬t2 ∧ ¬t1 ∧ obc ∧ ca ∧ cb), pat},
 {{(¬t2 ∧ ¬t1 ∧ obc ∧ oab ∧ ca), pat},
 {{(¬t2 ∧ ¬t1 ∧ ca ∧ cb ∧ cc), pbc},
```

$$\langle (\neg t_2 \wedge \neg t_1 \wedge oba \wedge cb \wedge cc), pbc \rangle .$$

According to this program, if b is on c , a and b are clear, and it is not time 2, then the agent can put a on b ; otherwise, if it is neither time 1 nor time 2, the agent can do a variety of other things. For instance, if b is on c and a and b are clear, the agent can put a on the table. This illustrates the generality of the program. Because it is not yet time 1, it is acceptable to undo progress (we might have some other reason for wanting to do this), because there is time to put a back on b before time 2. Notice that this program is not complete. There are situations for which it has no action, because there are block configurations that cannot be made to satisfy the goal in two actions. Notice also that, because this is a program of the standard form used by Gapps, it can be conjoined in with programs arising from other goals, such as global maintenance goals. Its generality, in allowing any sequence of actions that achieves the first condition before the second, makes it more likely that conjoining it in with a program expressing some other constraint will result in a non-null program.

3.2 Working In Parallel With an Anytime Planner

When the size of the state space is so large that doing exhaustive planning at compile time is impractical, it is possible to solve problems described as planning problems by integrating a run-time planning system with the Gapps framework.

We can express the planning process as an incremental computation, one step of which is done on each tick. On each tick the process generates an output, but it may be one that means "I don't have an answer yet." After some number of ticks, depending on the size of the planning problem, the planner will generate a real result. This result could be cached and executed as in a traditional system, or the agent could just take the first action and wait for the planner to generate a new plan.

Because time may have passed since the planner began its task, we must take care that the plan it generates is appropriate for the situation the agent finds itself in when the planner is finished. This can be guaranteed if the planner monitors the conditions in the world upon which the correctness of its plan depends. If any of these conditions becomes false, the planner can begin again. This behavior will be correct, though not always optimal. In the worst case, the planner will continuously emit the "I don't know" output and the agent will react reflexively to its environment without the benefit of a plan.

The kind of planner discussed above is a degenerate form of an anytime algorithm [3]. An anytime algorithm always has an answer, but the answer improves over time. In the example given above, the answer is useless for a while, then improves dramatically in one step. It might be useful to have planning algorithms that improve more gradually. Such algorithms exist for certain kinds of path planning, for instance, in which some path is returned at the beginning, but the algorithm works to make the path shorter or more efficient. There is

still a difficult decision to be made, however, about whether to take the first step of a plan that is known to be non-optimal or to spend more time planning. For many everyday activities, optimality is not crucial, and it will be sufficient to act on the basis of a simple plan, if a plan is required at all.

From the perspective of Gapps, the anytime planner is just a perceptual process that has state. It is "perceiving" conditions of the form: "the world is in a state such that if I do action α followed by action β , followed by action γ , my goal will be achieved." The following Gapps program makes use of such a planner, but also has the potential for reacting to emergency situations:

```
(defgoalr (ach (in room) [r t])
  (if (know-plan-for-getting-to-room r t)
    (ach execute-first-step
      (plan-for-getting-to-room r t))
    (if (time-is-critical-for-getting-to-room r t)
      (ach drive-in-the-direction-of-room r)
      (maint sit-still)))) .
```

If the agent has the goal of being in room r at time t , and he knows a plan for getting there, then he should execute the first step of that plan; otherwise, if it looks like time is running out, the agent should do the best action he can think of at the moment; if there is no problem with time, his best course of action is to sit still and wait until the perception component has produced a plan. These issues of combining planning and reactive action are explored more fully by Kaelbling [4].

3.3 Run-Time Goals

So far, we have only addressed the case in which the agent's top-level goal is specified at compile time. It will often be the case that it is useful to think of the agent as acquiring goals at run time. Before we can discuss ways of processing run-time goals, we must understand their semantics.

3.3.1 Dispatching

The simplest case of responding to run-time goals is to consider them to be another type of perceived information and write goal-reduction rules that are conditional on the given goal. As an example of this, an agent could be given the static compile-time goal of following orders and reduction rules of the following form:

```
(defgoalr (maint follow-orders)
  (if (current-request-pending)
    (ach goal-encoded-by (perceived-command))
    (do twiddle-thumbs)))

(defgoalr (ach goal-encoded-by params)
```

```

(if (move-command params)
  (ach do-move-command (get-destination params))
  (if (stop-command params)
    (ach stopped)
    ...)))

```

The agent will carry out requests as it perceives them by dispatching to the right goal-reductions based on the nature of the request. This method is sufficient for many cases, but requires the run-time goals to be of a few limited types, because the different types must be tested for and dispatched to directly.

3.3.2 Run Time Goal Reduction

An alternative to explicit dispatching on the types of goals is to interpret Gapps-style goal-reduction rules at run time. An interpreter for Gapps is very similar to the evaluation procedure, except that the result at each step is a set of possible actions, rather than a set of condition-action pairs. This is because the interpretation is taking place at run time, which allows all of the conditions to be evaluated during the interpretation process, rather than combined into a program that is to be evaluated later. Any action can be chosen from the set resulting from interpreting the top-level goal in the current situation.

Given a reduction-rule set Γ , we define the interpretation procedure as follows:

```

define interp(G)
case first(G)
do : make-action-set(second(G),rex-eval(third(G)))
and: conjoin-action-sets(interp(second(G)),interp(third(G)))
or : disjoin-action-sets(interp(second(G)),interp(third(G)))
not: interp(negate-goal-expr(second(G)))
if: if rex-eval(second(G)) then
    interp(third(G)) else
    interp(fourth(G))
maint,
ach: for all R in  $\Gamma$  such that match(G,head(R))
    disjoin-action-sets(interp(body(R))

```

The function `make-action-vector` takes an index and a value and returns the singleton set containing the action vector with the field specified by the index set to the indicated value. That is,

$$\text{make-action-vector}(i, v) = \{ \langle 0, \dots, v, \dots, 0 \rangle \}.$$

The value is calculated by evaluating, in the current state of the world, the Rex expression specifying the primitive action. Using the functions `mergeable` and `merge` described in Section 2.2, the conjunction of action sets can be defined as

$$\text{conjoin-action-sets}(A', A'') = \{ \text{merge}(a'_i, a''_j) \mid \text{mergeable}(a'_i, a''_j) \}$$

for $1 \leq i \leq m, 1 \leq j \leq n$ where

$$\begin{aligned} A' &= \{a'_1, \dots, a'_m\} \\ A'' &= \{a''_1, \dots, a''_n\}. \end{aligned}$$

The disjunction of two action sets is simply the union of the sets:

$$\text{disjoin-action-sets}(A', A'') = A' \cup A''.$$

The crucial difference between the interpretation procedure and the evaluation procedure is in the *if* case. When the interpreter encounters an *if* goal, it can simply test the condition in the current state of the world and go on to interpret the subgoal corresponding to the result of the test. This obviates the need for manipulating formal descriptions of conditions during the goal-interpretation process.

If the rule set is fixed at compile time and is not recursive, interpretation can be done by a fixed circuit (written, perhaps, in Rex) whose depth is equal to the length of the maximum-length chain of rules in the rule set. If the rule set is recursive, a depth bound will have to be imposed in order to guarantee real-time response. Another possibility would be to make this into an anytime algorithm by using iterative deepening search over the course of a number of ticks, and being careful that conditions that have already been evaluated do not change their values during the search process.

If the agent acquires goal reduction rules at run time, perhaps through learning, then the interpretation process can be carried out by general-purpose goal-reduction machinery. It can either be done in real time by a fixed circuit or over time by an anytime search procedure. If interpretation is to happen in real time, there must be a limit on the number of reduction rules that can be applied, in order to make the circuitry be of fixed size.

Conclusions

The Gapps goal-reduction formalism provides a flexible, declarative method for describing the action component of agents that must operate in real-time in dynamic worlds. It has a formal semantic grounding and has been implemented and used in a variety of robotic applications. In addition, it can be extended in a number of ways for use in domains with different types of complexity.

References

- [1] Isaac Asimov. *I, Robot*. Fawcett Crest, New York, New York, 1950.
- [2] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical Report AIM-864, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1985.

- [3] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988.
- [4] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning About Actions and Plans*, pages 395-410. Morgan Kaufmann, 1987.
- [5] Leslie Pack Kaelbling. Rex: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts, 1987.
- [6] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988.
- [7] Leslie Pack Kaelbling and Nathan J. Wilson. Rex programmer's manual. Technical Report 381R, Artificial Intelligence Center, SRI International, Menlo Park, California, 1988.
- [8] Stanley J. Rosenschein. Plan synthesis: A logical perspective. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, British Columbia, 1981.
- [9] Stanley J. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3(4):345-357, 1985.
- [10] Stanley J. Rosenschein. Synthesizing information-tracking automata from environment descriptions. In *Proceedings of Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada, 1989.
- [11] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In Joseph Halpern, editor, *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83-98. Morgan Kaufmann, 1986. An updated version appears as Technical Note 412, Artificial Intelligence Center, SRI International, Menlo Park, California.
- [12] Stanley J. Rosenschein and Leslie Pack Kaelbling. Integrating planning and reactive control. In *Proceedings of NASA/JPL Conference on Space Telerobotics*, Pasadena, California, 1989.
- [13] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1039-1046, Milan, 1987. Morgan Kaufmann.

C Foundations of Learning in Autonomous Agents

Foundations of Learning in Autonomous Agents

Leslie Pack Kaelbling*
Teleos Research
and
Stanford University

October 13, 1989

1 Introduction

Autonomous agents must learn to act in complex, noisy domains. This paper will provide a formal description of the problem of building autonomous agents that learn to act and will provide metrics for comparing learning algorithms that are appropriate for autonomous agents.

Why should we build learning agents? A program that "learns" is not intrinsically better than one that does not. One reason to build learning agents is that it is very difficult for humans to write explicit programs for agents that must work in complex, uncertain environments. In programming robots, for instance, it is common for a human programmer to learn a great deal about the operation of the robot's sensors and effectors in the course of debugging programs for the robot. It would be much easier and less time-consuming if the programmer were able to articulate only general principles about the environment, allowing the robot to experiment and learn about its own sensors and effectors. Another reason for building agents that learn to act is that we would like to have agents that are flexible enough to work in a variety of environments, adapting their perception and action strategies to the world in which they find themselves. Even if a human could completely specify the program for an agent to operate in a particular environment, the agent would have to be completely reprogrammed to move it to a new environment.

In these cases, the goal of the agent's designer is to have the agent learn what actions it should perform in which situations in order to maximize an external measure of success. All of the information the agent has about the external world

*This work was supported in part by a gift from the System Development Foundation and in part by the Air Force Office of Scientific Research under contract #F49620-89-C-0055.

is contained in a series of inputs that it receives from the environment. These inputs may encode information ranging from the output of a vision system to a robot's current battery voltage. The agent can be in many different states of information about the environment, and it must map each of these information states, or situations, to a particular action that it can perform in the world. The agent's mapping from situations to actions is referred to as an *action map*. Part of the agent's input from the world encodes the agent's *reinforcement*, which is a measure of how well the agent is performing in the world. The agent should learn to act in such a way as to maximize its total reinforcement.

As a concrete example, consider a simple robot with two wheels and two photo-sensors. It can execute five different actions: stop, go forward, go backward, turn left, and turn right. It can sense three different states of the world: the light in the left eye is brighter than that in the right eye, the light in the right eye is brighter than that in the left eye, and the light in both eyes is roughly equally bright. Additionally, the robot is given high values of reinforcement when the average value of light in the two eyes is increased from the previous instant. In order to maximize its reinforcement, this robot should turn left when the light in its left eye is brighter, turn right when the light in its right eye is brighter, and move forward when the light in both eyes is equal. The problem of learning to act is to discover such a mapping from information states to actions.

Thus, the problem of learning to act can be cast as a function-learning problem: the agent must learn a mapping from the situations in which it finds itself to the actions it can perform. In the simplest case, the mapping will be a pure function, but in general it can have state, allowing the action taken at a particular time to depend on any previous situation. In the past few years there has been a great deal of work in the artificial intelligence and theoretical computer science communities on the problem of learning pure Boolean-valued functions [10]. Unfortunately, this work is not directly relevant to the problem of learning action maps because of the different settings of the problem. In the traditional function-learning work, a learning algorithm is presented with a set or series of input-output pairs that specify the correct output to be generated for that particular input. This setting allows for effective function learning, but does not mirror the situation of an agent trying to learn an action map. The agent, finding itself in a particular input situation, must generate an action. It then receives a reinforcement value from the environment, indicating how effective that action was. The agent cannot, however, deduce the reinforcement value that would have resulted from executing any of its other actions. Also, if the environment is noisy, as it will be in general, just one instance of performing an action in a situation may not give an accurate picture of the reinforcement value of that action.

The problem of learning action maps by trial and error is often referred to as *reinforcement learning* because of its similarity to models used in psychological studies of behavior-learning in humans and animals. It can also be classified as *unsupervised learning* because correct answers are not provided by a teacher

[14]. One of the most interesting facets of the reinforcement learning problem is the tension between performing actions that are not well understood in order to gain information about their reinforcement value and performing actions that are expected to be good in order to increase overall reinforcement. If an agent knows that a particular action works well in a certain situation, it must trade off performing that action against performing another one that it knows nothing about, in case the second action is even better than the first. Another important aspect of the reinforcement-learning problem is that the actions that an agent performs influence the input situations in which it will find itself in the future. Rather than receiving an independently chosen set of input-output pairs, the agent has some control over what inputs it will receive and complete control over what outputs will be generated in response. In addition to making it difficult to make distributional statements about the inputs to the agent, this makes it possible for what seem like small "experiments" to cause the agent to discover an entirely new part of its environment.

Because of these differences in the setting of the learning task, algorithms (such as Michalski's *star* method [13], Mitchell's version spaces [15,16] and Valiant's algorithm for learning k -dnf [25]) and evaluation metrics (such as PAC-learning [24] and mistake bounds [12]) developed for traditional function learning are not appropriate for learning to act. This paper focuses on building formal foundations for the problem of learning in autonomous agents. These foundations must allow a clear statement of the problem and provide a basis for evaluating and comparing learning algorithms. It is important to establish such a basis: there are many instances [22,9] in the machine learning literature of people doing interesting work on learning agents, but reporting the results in a way that makes it difficult to compare them with the results of others.

2 Acting in a Complex World

An autonomous agent can be seen as acting in a world, continually executing a function that maps the agent's perceptual inputs to its effector outputs. Its world, or environment, is everything that is outside the agent itself, possibly including other robotic agents or humans. The agent operates in a cycle, receiving an input from the world, doing some computation, then generating an output that affects the world. The mapping that it uses may have state or memory, allowing its action at any time to depend, potentially, on the entire stream of inputs that it has received until that time. Such a mapping from an input stream to an output stream is referred to as a *behavior*.

In order to study the effectiveness of particular behaviors, whether or not they involve learning, we must model the connection between agent and world, understanding how an agent's actions affect its world and, hence, its own input stream.

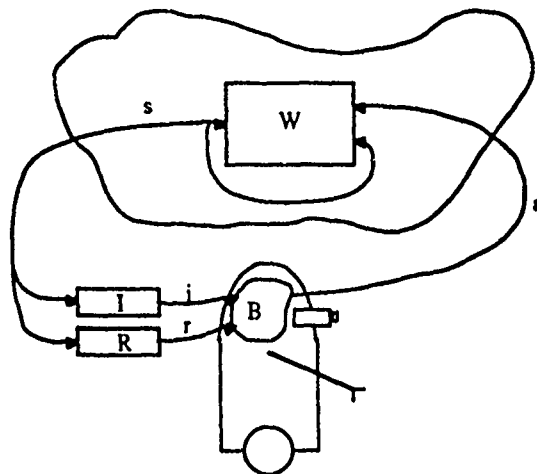


Figure 1: An agent's interaction with its world.

2.1 Modeling an Agent's Interaction with the World

The world can be modeled as a deterministic finite automaton whose state transitions depend on the actions of an agent. This model will be extended to include non-deterministic worlds in the next section. A world can be formally modeled as the triple (S, \mathcal{A}, W) , in which S is the set of possible states of the world, \mathcal{A} is the set of possible outputs from the agent to the world (or actions that can be performed by the agent), and W is the state transition function, mapping $S \times \mathcal{A}$ into S . Once the world has been fixed, the agent can be modeled as the 4-tuple (\mathcal{I}, I, R, B) where \mathcal{I} is the set of possible inputs from the world to the agent, I is a mapping from S to \mathcal{I} that determines which input the agent will receive when the world is in a given state, R is the reinforcement function of the agent that maps S into real numbers (it may also be useful to consider more limited models in which the output of the reinforcement function is Boolean-valued), and B is the behavior of the agent, mapping \mathcal{I}^* (streams of inputs) into \mathcal{A} . The expressions $i(t)$ and $a(t)$ will denote the input received by the agent at time t and the action taken by the agent at time t , respectively.

The process of an agent's interaction with the world is depicted in Figure 1. The world is in some internal state, s , which is projected into i and r by the input and reinforcement functions I and R . These values serve as inputs to the agent's behavior, B , which generates an action a as output. Once per synchronous cycle of this system, the value of a , together with the old value of world state s , is transformed into a new value of world state s by the world's transition function W .

Note that if the agent does not have a simple stimulus-response behavior, but

has some internal state, then the action taken by the behavior can be a function of both its input and its internal state. This internal state may allow the agent to discriminate among more states of the world and, hence, to obtain higher reinforcement values by performing more appropriate actions. To simplify the following discussion, actions will be conditioned only on the input, but the treatment is easily extended to the case in which the action depends on the agent's internal state as well.

2.2 Inconsistent Worlds

One of the most difficult problems that a learning agent must contend with is inconsistency. A world is said to be *inconsistent* for an agent if it is possible that, on two different occasions in which the agent receives the same input and generates the same action, the next states of the world differ in their reinforcement or the world changes state in such a way that the same string of future actions will have different reinforcement results. There are many different phenomena that can account for inconsistency:

- *The agent does not have the ability to discriminate among all world states.* If the agent's input function I is not one-to-one, which will be the case in general, then an individual input could have arisen from many world states. When some of those states respond differently to different actions, the world will appear inconsistent to the agent.
- *The agent has "faulty" sensors.* Some percentage of the time, the world is in a state s , which should cause the agent to receive $I(s)$ as input, but it appears that the world is in some other state s' , causing the agent to receive $I(s')$ as input instead. Along with the probability of error, the nature of the errors must be specified: are the erroneously perceived states chosen maliciously, or according to some distribution over the state space, or contingently upon what was to have been the correct input?
- *The agent has "faulty" effectors.* Some percentage of the time, the agent generates action a , but the world actually changes state as if the agent had generated a different action a' . As above, both the probability and nature of the errors must be specified.
- *The world has a probabilistic transition function.* In this case, the world is a stochastic automaton whose transition function, W' , actually maps $S \times A$ into a probability distribution over S (a mapping from S into the interval $[0, 1]$) that describes the probability that each of the states in S will be the next state of the world.

Some specific cases of noise phenomena above have been studied in the formal function-learning literature. Valiant [24] has explored a model of noise in which, with some small probability, the entire input instance to the agent can be chosen

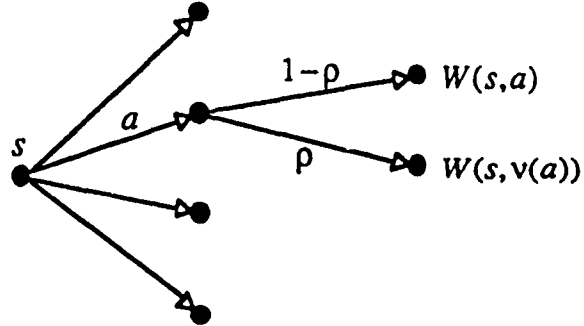


Figure 2: Modeling faulty effectors as a probabilistic world transition function.

maliciously. This corresponds, roughly, to having simultaneous faults in sensing and action that can be chosen in a way that is maximally bad for the learning algorithm. This model is overly pessimistic and is hard to justify in practical situations. Angluin [2] works with a model of noise in which input instances are misclassified with some probability; that is, the output part of an input-output pair is specified incorrectly. This is a more realistic model of noise, but is not directly applicable to the action-learning problem under consideration here.

If the behavior of faulty sensors and effectors is not malicious, each of the types of inconsistency discussed above can be described by transforming the original world model into one in which the set of world states, S , is identical to the set of agent inputs, \mathcal{I} , and in which the world has a probabilistic transition function. Reducing each of these phenomena to probabilistic world-transition functions allows the rest of the discussion of embedded behaviors to ignore the other possible modes of inconsistency. The remainder of this section shows how to transform worlds with each type of inconsistency into worlds with state set \mathcal{I} and probabilistic transition functions.

Consider an agent, embedded in a world with deterministic transition function W , whose effectors are faulty with probability ρ , so that when the intended action is a , the actual action is $\nu(a)$. This agent's situation can be described by a probabilistic world transition function $W'(s, a)$ that maps the value of $W(s, a)$ to the probability value $1 - \rho$, the value of $W(s, \nu(a))$ to the probability value ρ and all other states to probability value 0. That is,

$$\begin{aligned} W'(s, a)(W(s, a)) &= 1 - \rho \\ W'(s, a)(W(s, \nu(a))) &= \rho \end{aligned}$$

The result of performing action a in state s will be $W(s, a)$ with probability $1 - \rho$, and $W(s, \nu(a))$ with probability ρ . Figure 2 depicts this transition function. First, a deterministic transition is made based on the action of the agent; then, a probabilistic transition is made by the world. This model can be easily extended

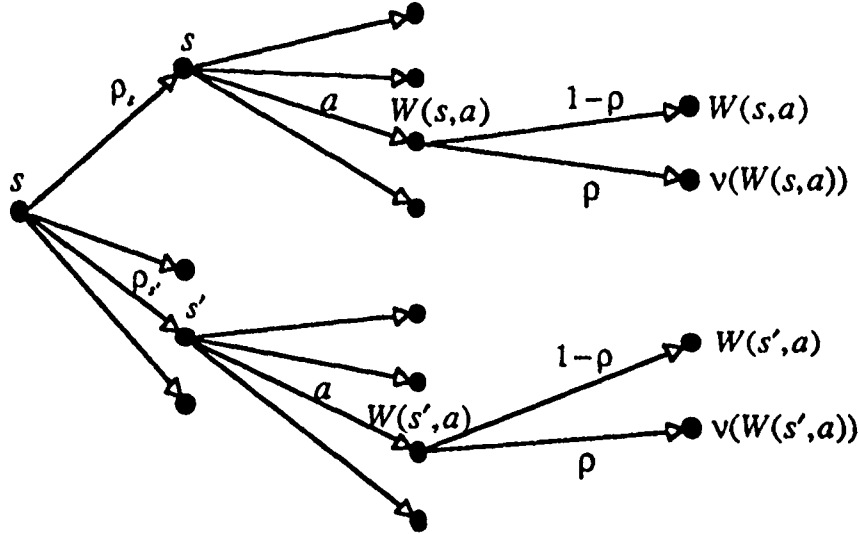


Figure 3: Modeling faulty sensors with multiple probabilistic transitions.

if ν is a mapping from actions to probability distributions over actions. In that case, for all a' not equal to a , the value of $W(s, a')$ is mapped to the probability value $\rho \nu(a)(a')$, which is the probability of an error, ρ , times the probability that action a' will be executed given that the agent intended to execute the action a . The value of $W(s, a)$ is mapped to the probability value $1 - \rho + \rho \nu(a)(a)$, which is the probability that there is no error, plus the probability that the error actually maps back to the correct action.

Faulty input sensors are somewhat more difficult to model. Let the agent's sensors be faulty with probability ρ , yielding a value $I(\nu(s))$ rather than $I(s)$. It is possible to construct a new model with a probabilistic world-transition function in which the states of the world are those that the agent *thinks* it is in. The model can be most simply viewed if the world makes more than one probabilistic transition, as shown in Figure 3. If it appears that the world is in state s , then with probability ρ_s , it actually is, and the first transition is to the same state. The rest of the probability mass is distributed over the other states in the inverse image of s under ν , $\nu^{-1}(s)$, causing a transition to some world state s' with probability ρ_s' . Next, there is a transition to a new state on the basis of the agent's action according to the original transition function W . Finally, with probability ρ , the world makes a transition to the state $\nu(W(s', a))$, allowing for the chance that this result will be misperceived on the next tick. In Figure 4, this diagram is converted into a more standard one, in which the agent performs an action, then the world makes a probabilistic transition. This construction can also be extended to the cases in which $\nu(s)$ is a probability distribution over

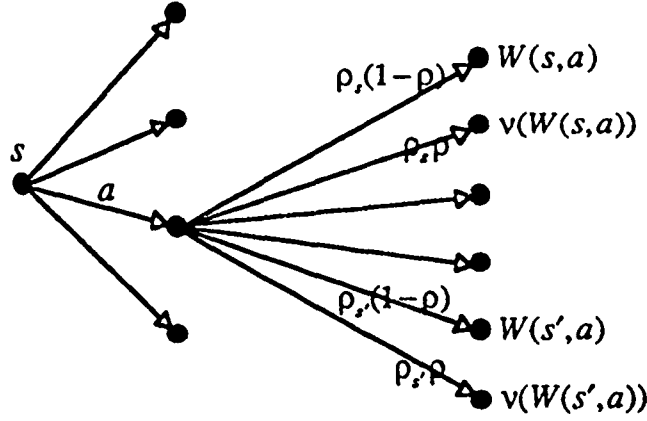


Figure 4: Modeling faulty sensors as a probabilistic world transition function.

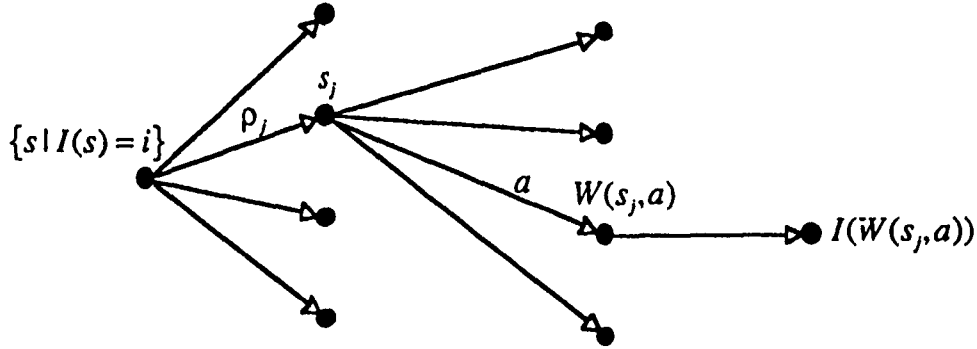


Figure 5: Modeling inability to discriminate among worlds.

S and in which the initial world-transition function is probabilistic.

To model an agent's inability to discriminate among worlds, it is possible to construct a new model of the world in which the elements of \mathcal{I} are the states, standing for equivalence classes of the states in the old model. Let $\{s_1, \dots, s_n\}$ be the inverse image of i under I . There is a probabilistic transition to each of the s_j , based on the probability, ρ_j , that the world is in state s_j given that the agent received the input i . From each of these states, the world makes a transition on the basis of the agent's action, a , to the state $W(s_j, a)$, which is finally mapped back down to the new state space by the function I . This process is depicted in Figure 5 and the resulting transition function is shown in Figure 6.

In the construction for faulty sensors, it is necessary to evaluate the probability that the world is in some state s_k , given that it appears to the agent to

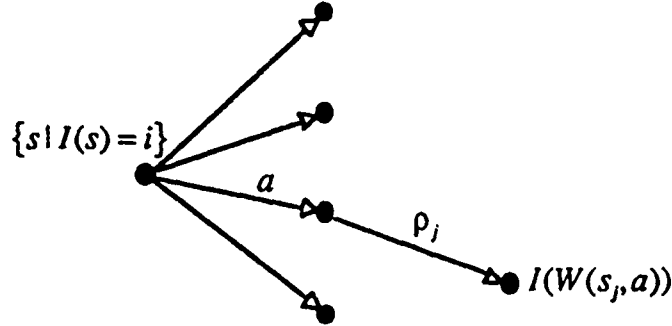


Figure 6: Modeling inability to discriminate among worlds as a probabilistic world transition function.

be in another state s . This probability depends on the unconditional probability that the world is in the state s_k , as well as the unconditional probability that the world appears to be in the state s . These unconditional probabilities depend, in the general case, on the behavior that the agent is executing, so the construction cannot be carried out before the behavior is fixed. A similar problem exists for the case of lack of discrimination: it is necessary to evaluate the probability that the world is in each of the individual states in the inverse image of input i under I given that the agent has input i . These probabilities also depend on the behavior that is being executed by the agent. This leads to a very complex optimization problem that is, in its general form, beyond the scope of this paper.

The rest of the paper will be concerned only with worlds that are globally consistent for the learning agent. A world is *globally consistent* for an agent if and only if for all inputs $i \in \mathcal{I}$ and actions $a \in A$, the *expected* value of the reinforcement given i and a is constant. Global consistency allows for variations in the result of performing an action in a situation, as long as the expected, or average, result is the same. It simply requires that there not be variations in the world that are undetectable by the agent and that affect its choice of action. If the transformation described above has been carried out so that the sets \mathcal{I} and S are the same, this is tantamount to requiring that the world be a Markov decision process with stationary transition and output probabilities [11]. In addition, the following discussion will assume that the world is consistent over changes in the agent's behavior.

2.3 Learning Behaviors

The problem of programming an agent to behave correctly in a world is to choose some behavior B , given that the rest of the parameters of the agent and world are fixed. If the programmer does not know everything about the world, or if he

```

s := s0
loop
  i := input
  a := e(s,i)
  output a
  r := reinforcement
  s := u(s,i,a,r)
end loop

```

Figure 7: General algorithm for learning behaviors.

wishes the agent he is designing to be able to operate in a number of different worlds, he must program an agent that will *learn to behave correctly*. That is, he must find a behavior B' that, through changing parts of its internal state on the basis of its perceptual stream, eventually converges to some behavior B'' that is correct for the world that gave rise to its perceptions. Of course, to say that a program learns is just to take a particular perspective on a program with internal state. A behavior with state can be seen as "learning" if parts of its state eventually converge to some fixed or slowly-varying values. The behavior that results from those parameters having been fixed in that way can be called the "learned behavior."

A *learning behavior* is an algorithm that learns an appropriate behavior for an agent in a world. It is itself a behavior, mapping elements of \mathcal{I} to elements of \mathcal{A} , but it requires the additional input $R(s)$ for every state s , in order to know the reinforcement value of the state for the agent. A learning behavior consists of three parts: an initial state s_0 , an update function u , and an evaluation function e . At any moment, the internal state encodes whatever information the learner has chosen to save about its interactions with the world. The update function maps an internal state of the learner, an input, an action, and a reinforcement value into a new internal state, adjusting the current state based on the reinforcement resulting from performing that action in that input situation. The evaluation function maps an internal state and an input into an action, choosing the action that seems most useful for the agent in that situation, based on the information about the world stored in the internal state. Recall that an action can be useful for an agent either because it has a high reinforcement value or because the agent knows little about its outcome.

A general algorithm for learning behaviors, based on these three components, is shown in Figure 7. The internal state is initialized to s_0 , then the algorithm loops forever. An input is read from the world and the evaluation function is applied to the internal state and the input, resulting in an action, which is then output. At this point, the world changes to a new state. The program next determines the reinforcement associated with the new situation, uses that information, together with the last input and action, to update the internal

state, then goes back to the top of its loop. Formulating learning behaviors in terms of s_0 , c , and u facilitates building experimental frameworks that allow testing of different learning behaviors in many different worlds.

3 Performance Criteria

In order to compare algorithms for learning behaviors, we must fix the criteria on which they are to be judged. There are three major considerations: correctness, convergence, and time-space complexity. First, we must determine the correct behavior for an agent in a domain. Then we can measure to what degree a learned behavior approximates the correct behavior and the speed, in terms of the number of interactions with the world, with which it converges. We must also be concerned with the amount of time and space needed for computing the update and evaluation functions and with the size of the internal state of the algorithm.

As well as comparing the performance of different algorithms for a particular world, it is useful to study the way different performance measures of an algorithm vary as a function of independent variables that characterize a world. Such independent variables might include: the sizes of \mathcal{I} and \mathcal{A} and the values of the performance measures on the random algorithm (one that chooses among the available actions randomly at each time step). These kinds of comparisons are not pursued further in this paper, but are enabled once objective performance criteria are chosen.

3.1 Correctness

When shall we say that a behavior is correct for an agent in an environment? There are many possible answers that will lead to different learning algorithms and analyses. An important quantity is the expected reinforcement that the agent will receive in the next instant, given that the current input is $i(t)$ and the current action is $a(t)$, which can be expressed as

$$\begin{aligned} er(i(t), a(t)) &= E(R(i(t+1)) \mid i(t), a(t)) \\ &= \sum_{i' \in \mathcal{I}} R(i') W'(i(t), a(t))(i'). \end{aligned}$$

It is the sum, over all possible next states, of the probability that the world will make a transition to that state times its reinforcement value. This formulation assumes that the inputs directly correspond to the states of the world and that W' is a probabilistic transition function. If the world is globally consistent for the agent, the process is Markov and the times are irrelevant in the above definition, allowing it to be restated as

$$er(i, a) = \sum_{i' \in \mathcal{I}} R(i') W(i, a)(i').$$

One of the simplest criteria is that a behavior is correct if, at each step, it does the action that is expected to cause the most reinforcement to be received on the next step. A correct behavior, in this case, is one that generates actions that are optimal under the following definition:

$$\forall i \in \mathcal{I}, a \in A. \text{Opt}(i, a) \leftrightarrow \forall a' \in A. er(i, a) \geq er(i, a') .$$

Optimal behavior is defined as a relation on inputs and actions rather than as a function, because there may be many actions that are equally good for a given input. However, it can be made into a function by breaking ties arbitrarily. This is a local criterion that may cause the agent to sacrifice promises of future reinforcement for immediately attainable current reinforcement.

The concept of expected reinforcement can be made more global by considering the total expected reinforcement for a finite future interval, or *horizon*, given that an action was taken in a particular input situation. This is often termed the *value* of an action, and it is computed with respect to a particular behavior (because the value of the next action taken depends crucially on how the agent will behave after that). In the following, expected reinforcement is computed under the assumption that the agent will act optimally the rest of the time. The expected reinforcement, with horizon k , of doing action a in input situation i at time t is defined as

$$er_k(i(t), a(t)) = E\left(\sum_{j=1}^k R(i(t+j)) \mid i(t), a(t), \forall h < k. \text{Opt}_{k-h}(i(t+h), a(t+h))\right) .$$

This expression can be simplified to a recursive, time-independent formulation, in which the k -step value of an action in a state is just the one-step value of the action in the state plus the $k-1$ -step value of the optimal action in the following state:

$$er_k(i, t) = er(i, a) + \sum_{i' \in \mathcal{I}} W'(i, a)(i') er_{k-1}(i', \text{Opt}_{k-1}(i')) .$$

This definition is recursively dependent on the definition of optimality k steps into the future, Opt_k :

$$\forall i \in \mathcal{I}, a \in A. \text{Opt}_k(i, a) \leftrightarrow \forall a' \in A. er_k(i, a) \geq er_k(i, a') .$$

The values of er_1 and Opt_1 are just er and Opt given above. The k -step value of action a in situation i at time t , $er_k(i, a)$, can be computed by dynamic programming. First, the Opt_1 relation is computed; this allows the er_2 function to be calculated for all i and a . Proceeding for k steps will generate the value for er_k . Because of the assumption that the world is Markov, these values are not dependent on the time. However, if k is large, the computational expense of this method is prohibitive.

Another way to define global optimality is to consider an infinite sum of future reinforcement values in which more recent values are weighted more heavily

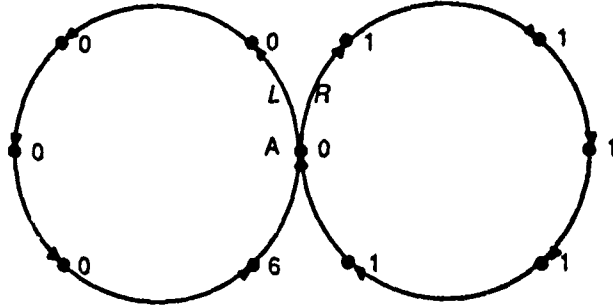


Figure 8: A sample deterministic world.

than older values. This is referred to as a *discounted sum*, depending on the parameter γ to specify the rate of discounting. *Expected discounted reinforcement* at time t is defined as

$$er_{\gamma}(i(t), a(t)) = E\left(\sum_{j=1}^{\infty} \gamma^{j-1} R(i(t+j)) \mid i(t), a(t), \forall h. \text{Opt}_{\gamma}(i(t+h), a(t+h))\right).$$

Properties of the exponential allow us to reduce this expression to

$$er(i(t), a(t)) + \gamma er_{\gamma}(i(t+1), a(t+1)),$$

which can be expressed independent of time as

$$er_{\gamma}(i, a) = er(i, a) + \gamma \sum_{i' \in \mathcal{I}} W'(i, a)(i') er_{\gamma}(i', \text{Opt}_{\gamma}(i')).$$

The related definition of γ -discounted optimality is given by

$$\forall i \in \mathcal{I}, a \in A. \text{Opt}_{\gamma}(i, a) \leftrightarrow \forall a' \in A. er_{\gamma}(i, a) \geq er_{\gamma}(i, a').$$

For a given value of γ and a proposed definition of Opt_{γ} , er_{γ} can be found by solving a system of equations, one for each possible instantiation of its arguments. A dynamic programming method called *policy iteration* [20] can be used in conjunction with that solution method to adjust policy Opt_{γ} until it is truly the optimal behavior. This definition of optimality is more widely used than finite-horizon optimality because its exponential form makes it more computationally tractable. It is also an intuitively satisfying model, with slowly diminishing importance attached to events in the distant future.

As an illustration of these different measures of optimality, consider the world depicted in Figure 8. In state 0, the agent has a choice as to whether to go right or left; in all other states the world transition is the same no matter what the agent does. In the left loop, the only reinforcement comes at the last state before

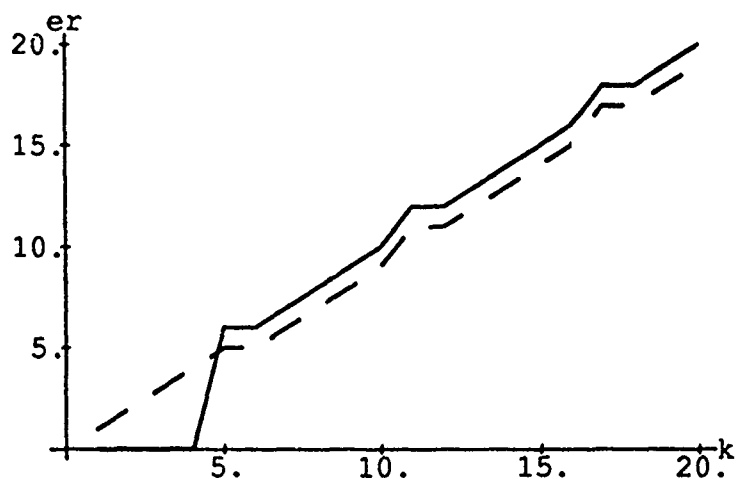


Figure 9: Plot of expected return against horizon k . Solid line indicates strategy of going left first, then behaving optimally. Dashed line indicates strategy of going right first, then behaving optimally.

state 0, but it has value 6. In the right loop, each state has reinforcement value 1. Thus, the average reinforcement is higher around the left loop, but it comes sooner around the right loop. The agent must decide what action to take in state 0. Different definitions of optimality lead to different choices of optimal action.

Under the local definition of optimality, we have $er(0, L) = 0$ and $er(0, R) = 1$. The expected return of going left is 0 and of going right is 1, so the optimal action would be to go right.

Using the finite horizon definition of optimality, which action is optimal depends on the horizon. For very short horizons, it is clearly better to go right. When the horizon, k , is 5, it becomes better to go left. A general rule for optimal behavior is that when in state 0, if the horizon is 5 or more, go left, otherwise go right. Figure 9 shows a plot of the values of going left (solid line) and going right (dashed line) initially, assuming that all choices are made optimally thereafter. We can see that going right is initially best, but it is dominated by going left for all $k \geq 5$.

Finally, we can consider discounted expected value. Figure 10 shows a plot of the values of the strategies of always going left at state 0 (solid line) and always going right at state 0 (dashed line) plotted as a function of γ . When there is a great deal of discounting (γ is small), it is best to go right because the reward happens sooner. As γ increases, going left becomes better, and at approximately $\gamma = 0.15$, going left dominates going right.

One way to design learning behaviors that have these difficult kinds of global

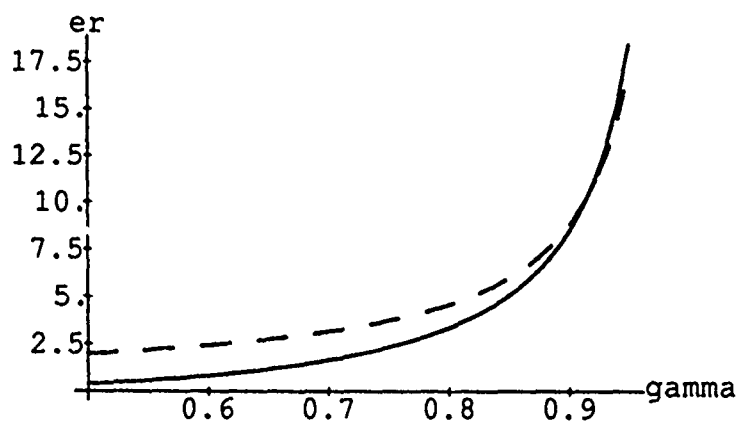


Figure 10: Plot of expected return against discount factor γ . Solid line indicates strategy of always going left. Dashed line indicates strategy of always going right.

optimality is to divide the problem into two parts: transducing the global reinforcement signal into a local reinforcement signal and learning to perform the locally best action. The global reinforcement signal is the stream of values of $R(i(t))$ that come from the environment. The optimal local reinforcement signal, $\bar{R}(i(t))$, can be defined as $R(i(t)) + \gamma \text{er}_{\gamma}(i(t), \text{Opt}_{\gamma}(i(t)))$. It is the value of the state $i(t)$ assuming that the agent acts optimally. As shown by Sutton [22], this signal can be approximated by the value of the state $i(t)$ given that the agent acts how it is currently acting. Sutton's temporal difference (TD) algorithm provides a way of learning to generate the local reinforcement signal from the global reinforcement signal in such a way that, if combined with a correct local learning algorithm, it will converge to the true optimal local reinforcement values [22,23]. A complication introduced by this method is that, from the local behavior-learner's point of view, the world is not stationary. This is because it takes time for the TD algorithm to converge and because changes in the behavior cause changes in the values of states and therefore in the local reinforcement function.

The following discussion will be in terms of some definition of the optimality of an action for a situation, $\text{Opt}(t, a)$, which can be defined in any of the three ways above, or in some novel way that is more appropriate for the domain in which a particular agent is working.

3.2 Convergence

Correctness is a binary criterion: either a behavior is or is not correct for its world. Since correctness requires that the behavior perform the optimal actions from the outset, it is unlikely that any "learning" behavior will ever be correct. Using a definition of correctness as a reference, however, it is possible to develop other measures of how close particular behaviors come to the optimal behavior. This section will consider two different classes of methods for characterizing how good or useful a behavior is in terms of its relation to the optimal behavior.

3.2.1 Classical Convergence Measures

Early work in the theory of machine learning [5,7] was largely concerned with *learning in the limit*. Researchers were interested in characterizing whether or not a learning strategy would converge to the correct behavior in the limit. A behavior converges to the optimal behavior in the limit if there is some time after which every action taken by the behavior is the same as the action that would have been taken by the optimal behavior. Work in learning-automata theory has relaxed the requirements of learning in the limit by applying different definitions of probabilistic convergence to the sequence of internal states of a learning automaton [18].

An important recent development in this area is a model of Boolean-function learning algorithms that are *probably approximately correct* (PAC) [24,2], that is, that have a high probability of converging to a function that closely approximates the optimal function. The correctness of a function is measured with respect to a fixed probability distribution on the input instances—a function is said to approximate another function to degree ϵ if the probability that they will disagree on any instance chosen according to the given probability distribution is less than ϵ . This model requires that there be a fixed distribution over the input instances and that each input to the algorithm be drawn according to that distribution.

For an agent to act effectively in the world, its inputs must provide some information about the state that the world is in. In general, when the agent performs an action it will bring about a change in the state of the world and, hence, a change in the information the agent receives about the world. Thus, it will be very unlikely that such an agent's inputs could be modeled as being drawn from a fixed distribution, making PAC-convergence an inappropriate model for autonomous agents.

In addition, the PAC-learning model is distribution-independent—it seeks to make statements about the performance of algorithms no matter how the input instances are distributed. As Buntine has pointed out [6], its predictions are often overly conservative for situations in which there is *a priori* information about the distribution of the input instances, or in which something is known about the actual sample, such as how many distinct elements it contains.

3.2.2 Measuring Error over an Agent's Lifetime

None of the classical convergence measures take into account the behavior of the agent during the period in which it converges. Instead, they make what is, for an agent acting in the world, an artificial distinction between a learning phase and an acting phase. Autonomous agents that have extended run times will be expected to learn for their entire lifetime. Because they may not encounter certain parts or aspects of their environments until arbitrarily late in the run, it is inappropriate to require mistakes to be made before some fixed deadline.

Another way of characterizing the performance of a function-learning algorithm is to count the divergences it makes from the optimal function. Littlestone [12] has investigated this model extensively, characterizing the optimal number of 'mistakes' for a Boolean-function learner and presenting algorithms that perform very well on certain classes of Boolean functions. This model is intuitively pleasing, making no restrictive division into learning and acting phases, but it is not presented as being suited to noisy or inconsistent domains. However, by assimilating the inconsistency of the domain into the definition of the target function, as in the requirement for optimal behavior, Opt , we can make use of mistake bounds in inconsistent domains. A behavior is said to make an *avoidable mistake* if, given some input instance i , it generates action a and $\text{Opt}(i, a)$ does not hold; that is, there was some other action that would have had a higher expected reinforcement.

Avoidable mistake bounds take into account the fact that many mistakes cannot be avoided by an agent with limited sensory abilities and unreliable effectors. However, that measure is not entirely appropriate, because every non-optimal choice of action is considered to be a mistake of the same magnitude. The expected error of an action a given an input i , $\text{err}(a, i)$, is defined to be

$$\text{err}(a, i) = \text{er}(a', i) - \text{er}(a, i) ,$$

in which a' is any action such that $\text{opt}(a', i)$. The expected error associated with an optimal action is 0; for a non-optimal action, it is just the decrease in expected reinforcement due to having executed that action rather than an optimal one. The error of a behavior, either in the limit, or for runs of finite length, can be measured by summing the errors of the actions it generates. This value, referred to in the statistics literature as the *regret* of a strategy [4], represents the expected amount of reinforcement lost due to executing this behavior rather than an optimal one. This is an appropriate performance metric for agents embedded in inconsistent environments because it measures expected loss of reinforcement, which is precisely what we would like to minimize in our agents.

In many situations, the optimal behavior is unknown or difficult to compute, which makes it difficult to calculate the error of a given behavior. It is still possible to use this measure to compare two different behaviors for the same agent and environment. The expected reinforcement for an algorithm over some time period can be estimated by running it several times and averaging the

resulting total reinforcements. Because expectations are additive, the difference between the expected error of two algorithms is the same as the difference between their expected total reinforcement values. Thus, the difference between average reinforcements is a valid measure of comparison that provides a measure of a behavior's correctness that is independent of the internal architecture of the algorithm and that can be used to compare results across a wide variety of techniques.

3.3 Time and Space Complexity

Autonomous agents must operate in the real world, continually receiving inputs from and performing actions on their environment. Because the world changes dynamically, an autonomous agent must be *reactive*—always aware of and reacting to change in its environment. To ensure reactivity, an agent must operate in *real-time*; that is, its sense-compute-act cycle must keep pace with the unfolding of important events in the environment. The exact constraints on the reaction time of an agent are often difficult to articulate, but it is clear that, in general, unbounded computation must never take place.

A convenient way to guarantee real-time performance is to require that the behavior spend only a constant amount of time, referred to as a 'tick,' generating an action in response to each input. If the behavior is a learning behavior, the learning process must also spend only a constant amount of time on each input instance. There are two strategies for designing such a learning system: incremental and batch.

An incremental system processes each new data set or learning instance as it arrives as input. The processing must be efficient enough that the system is always ready for new data when it arrives. If new relevant data can arrive every tick, the learning algorithm must spend only one constant tick's worth of time on each instance. The requirement for incrementality can, theoretically, be relaxed to yield a batch system, in which a number of learning instances are collected, then processed for many ticks. As long as the learning system adheres to the tick discipline, this process need not interfere with the reactivity of the rest of the system. Working in batch mode may limit the usefulness of the learning system to some degree, however, because the system will be working with old data that may not reflect the current situation and it will force the data that arrives during the computation phase to be ignored. When using this method, the input data must be sampled with care, in order to avoid statistical distributions of inputs that do not reflect those of the external world.

An algorithm can be said to be *strictly incremental* if it uses a bounded amount of time and space throughout its entire lifetime. This is in contrast with such approaches as Kibler and Alia's instance-based learning [1], which is incremental in that it processes one instance at a time, but is not strictly incremental because instances are stored in a memory whose size may increase without bound. For an incremental system that processes one instance per tick

to perform in real time, it must be strictly incremental.

The amount of time an incremental behavior spends on each input should not vary as a function of the number of inputs that have been received. It will, however, depend on the size of the input and the output, but that is fixed at design time. This allows the programmer to know how long each tick of the learning behavior will take to compute on the available hardware and to compare that rate with the pace of events in the world. Any formalization of the interaction between an agent and its world will depend on the rate of the interaction; behaviors that work at rates different from the chosen one will essentially be working in a different environment. The expected values of optimal behaviors for different reaction rates will be quite different. In general, up to some minimum value, the faster an agent can interact with the world, the better (otherwise the agent does not have time to avert impending bad events), so we should strive for the most efficient algorithms possible, though a slow algorithm with better convergence properties might be preferable to a fast algorithm that was far from optimal.

Complex agents, such as mobile robots, with a wide variety of sensors and effectors will have a huge number of possible inputs and outputs. If algorithms for these agents are to be practical, they must have time and space complexity that is polynomial in the number of input bits, $\lg(|\mathcal{I}|)$, and the number of output bits, $\lg(|\mathcal{A}|)$, rather than the the number of inputs and outputs. This will probably only be achievable by limiting the class of behaviors that can be learned by the agent.

4 Related Work

The problem of learning the structure of a finite-state automaton from examples has been studied by many theoreticians, including Moore [17], Gold [8] and, more recently, Rivest and Schapire [19]. This is a very difficult problem that has only been studied in the case of deterministic automata. If the entire structure of the world can be learned, it is conceptually straightforward to compute the optimal behavior. It is important to note, however, that learning an action-map that maximizes reinforcement is not necessarily as complex as learning the world's transition function.

A number of different groups of researchers have considered the problem of designing algorithms for reinforcement learning and, in the process, have addressed the issue of measures for performance of reinforcement learning algorithms.

Statisticians have studied reinforcement learning in the guise of *k-armed bandit* problems, in which the agent has k possible actions and only reinforcement as input [4]. The work has primarily concerned the existence of optimal strategies given various kinds of *a priori* information about the possible distributions of the payoffs of the individual arms. The notion of *regret* was developed in the

context of choosing the optimal behavior in the minimax setting, in which the worst is assumed about the world. These strategies are, in general, computationally intractable and require, except in the minimax case, information that is unavailable in the current setting of the problem.

More appropriate to agents that must learn to behave in the world is the work of researchers in the field of *learning automata* [18]. They classify their algorithms according to whether they are expedient (better than the random strategy), optimal, or ϵ -optimal (some parameter can be chosen to make the behavior arbitrarily "close" to optimal). In addition, there are methods for characterizing the convergence rate for some learning-automata algorithms. These evaluation methods are tailored for the case in which the learning behavior's only internal state is a vector of probabilities, one for each possible action, that characterize the probability of the agent performing that action. Also, no consideration is made of the effectiveness of the algorithm during the time before it converges.

Watkins [27] presents a very clear discussion of different types of optimality from an operations-research perspective and characterizes possible algorithms for learning optimal behavior from delayed rewards. Williams [28] presents a theoretical view of a connectionist reinforcement-learning algorithm [3] as a form of gradient search. Sutton [22,23] shows how to divide the problem of learning from delayed reinforcement into the problems of locally optimal behavior learning and secondary reinforcement-signal learning.

5 Conclusion

This paper has studied the problem of building agents that learn about acting in complex, inconsistent environments. It has established local and global definitions of optimality of behaviors in non-deterministic worlds and has provided an implementation-independent measure of deviation from the optimal. This framework for the comparison of algorithms will allow researchers to develop new algorithms and compare them rigorously to one another.

A particularly interesting direction to pursue is how to make the algorithms more efficient in time and space and closer to optimal in behavior by making assumptions about the environment. Examples of this are Van de Velde's work on learning to optimize usefulness of results rather than their correctness [26] and Russell's use of determinations [21]. The only hope for the machine learning enterprise is that there are regularities in the world that will make efficient learning possible.

Acknowledgements

This paper is much improved thanks to insightful comments by Rich Sutton, Walter Van de Velde, and Stan Rosenschein. Thanks also to Guy Boy, Ann

Reid and Laura Wasylenki.

References

- [1] David W. Aha and Dennis Kibler. Noise tolerant instance-based learning algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 1, pages 794-799, Detroit, Michigan, 1989. Morgan Kaufmann.
- [2] Dana Angluin and Philip Laird. Learning from noisy examples. *Machine Learning*, 2(4):343-370, 1988.
- [3] A. G. Barto and P. Anandan. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360-374, 1985.
- [4] Donald A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London, 1985.
- [5] L. Blum and N. Blum. Towards a mathematical theory of inductive inference. *Information and Control*, 28:125-155, 1975.
- [6] Wray Buntine. A critique of the Valiant model. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 1, pages 837-842, Detroit, Michigan, 1989. Morgan Kaufmann.
- [7] E. Mark Gold. Language identification in the limit. *Information and Control*, 10:447-474, 1967.
- [8] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302-320, 1978.
- [9] John J. Grefenstette. Incremental learning of control strategies with genetic algorithms. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 340-344, Ithaca, New York, 1989. Morgan Kaufmann.
- [10] David Haussler. New theoretical directions in machine learning. *Machine Learning*, 2(4):281-284, 1988.
- [11] John G. Kemeny and J. Laurie Snell. *Finite Markov Chains*. Springer-Verlag, New York, 1976.
- [12] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning*, 2(4):245-318, 1988.

- [13] Ryszard S. Michalski. A theory and methodology of inductive learning. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 4. Tioga, 1983.
- [14] Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach*, volume 2. Morgan Kaufmann, Los Altos, California, 1986.
- [15] Tom M. Mitchell. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 305-310, Cambridge, Massachusetts, 1977.
- [16] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203-226, 1982.
- [17] Edward F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129-153. Princeton University Press, Princeton, New Jersey, 1956.
- [18] Kumpati Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction*. Prentice-Hall, Englewood, New Jersey, 1989.
- [19] Ronald L. Rivest and Robert E. Schapire. A new approach to unsupervised learning in deterministic environments. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 364-375, Irvine, California, 1987. Morgan Kaufmann.
- [20] Sheldon M. Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, New York, 1983.
- [21] Stuart J. Russell. Tree-structured bias. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 641-645, Minneapolis-St. Paul, Minnesota, 1988.
- [22] Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, Massachusetts, 1984.
- [23] Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9-44, 1988.
- [24] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134-1142, 1984.
- [25] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 560-566, Los Angeles, California, 1985. Morgan Kaufmann.

- [26] Walter Van de Velde. Quality of learning. In *Proceedings of the European Conference on Artificial Intelligence*, pages 408-413, Munich, 1988. Pitman Publishing.
- [27] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.
- [28] Ronald J. Williams. Reinforcement learning in connectionist networks: A mathematical analysis. Technical report, Institute for Cognitive Science, University of California, San Diego, La Jolla, California, 1986.

D Learning Functions in k -DNF from Reinforcement

Learning Functions in k -DNF from Reinforcement

Leslie Pack Kaelbling*
Telcos Research
and
Stanford University

Abstract

An agent that must learn to act in the world by trial and error faces the *reinforcement learning* problem, which is quite different from standard concept learning. Although good algorithms exist for this problem in the general case, they are quite inefficient. One strategy is to find restricted classes of action strategies that can be learned more efficiently. This paper pursues that strategy by developing algorithms that can efficiently learn action maps that are expressible in k -DNF. Both connectionist and classical statistics-based algorithms are presented, then compared empirically on three test problems. Modifications and extensions that will allow the algorithms to work in more complex domains are also discussed.

1 Reinforcement Learning

Consider an agent that must learn to act in the world. At each moment in time, it gets information about the world from its sensors and must choose an action to take. Having executed an action, the agent gets a signal from the world that indicates how well the agent is performing; we shall call this a *reinforcement* signal. The reinforcement signal can be binary or real-valued and it will typically be noisy.

This learning scenario is quite different from standard concept learning, in which a teacher presents the learner with a set of input/output pairs. In the reinforcement-learning scenario, the agent must choose an output to generate in response to each input. The reinforcement signal it receives indicates only how successful that output was; it carries no information about how successful other outputs might have been. In addition, the fact that the reinforcement signal is noisy means that each output will have to be generated a number of times in order for the agent to acquire an accurate picture of which is better. In reinforcement-learning situations, an agent may choose an action because it expects it to have good results; however,

*This work was supported by the Air Force Office of Scientific Research under contract F49620-89-C-0055.

it may also choose an action in order to gain information about its expected results. The tradeoff between acting to gain reinforcement and acting to gain information makes this problem especially interesting. The formal foundations of reinforcement learning have been widely studied [Kaelbling, 1989b, Kaelbling, 1989a, Narendra and Thathachar, 1989, Berry and Fristedt, 1985, Williams, 1986].

This paper will focus on a simple case of the reinforcement learning problem in which the following assumptions hold:

- the agent has only two possible actions
- the reinforcement signal at time $t + 1$ reflects only the success of the action taken at time t
- reinforcement received for performing a particular action in a particular situation is 1 with some probability p and 0 with probability $1 - p$ and each trial is independent
- the expected reinforcement value of doing a particular action in a particular input situation stays constant for the entire run of the learning algorithm

Section 6 discusses the extension of the results in this paper to situations in which each of the above assumptions is relaxed.

2 Complexity Versus Efficiency

There are a number of good algorithms for the reinforcement-learning scenario we are interested in, including learning-automata algorithms [Narendra and Thathachar, 1989], Sutton's reinforcement-comparison methods [Sutton, 1984], and Kaelbling's interval-estimation methods [Kaelbling, forthcoming]. These algorithms were originally developed for the case when the agent has no inputs other than reinforcement and merely needs to decide which action it should take all the time. They can be extended to the case of having many input situations simply by making a copy of the algorithm for each possible input situation. This method works well, but results in algorithms with space complexity proportional, at least, to the number of possible input situations. In addition, no generalization is exhibited; that is, the combined algorithms

do not take advantage of the common intuition that "similar" input situations are likely to require "similar" actions.

We can think of agents as learning *action maps*: mappings from input situations to actions. If an agent must be able to learn action maps of arbitrary complexity, then the methods described above are as good as any. However, if we restrict the class of action maps that we expect an agent to learn, we can invent algorithms for learning those maps that are much more efficient than algorithms for the general case.

A restriction that has proved useful to the concept-learning community is to the class of functions that can be expressed as propositional formulae in *k*-DNF. A formula is said to be in *disjunctive normal form* (DNF) if it is syntactically organized into a disjunction of purely conjunctive terms; there is a simple algorithmic method for converting any formula into DNF [Enderton, 1972]. A formula is in the class *k*-DNF if and only if its representation in DNF contains only conjunctive terms of length *k* or less. There is no restriction on the number of conjunctive terms—just their length. Whenever *k* is less than the number of atoms in the domain, the class *k*-DNF is a restriction on the class of functions.

Valiant was one of the first to consider the restriction to learning functions expressible in *k*-DNF [Valiant, 1984, Valiant, 1985]. He developed the following algorithm for learning functions in *k*-DNF from input-output pairs, which actually only uses the input-output pairs with output 0:

Let T be the set of conjunctive terms of length k over the set of atoms (corresponding to the input bits) and their negations and let L be the number of learning instances required to learn the concept to the desired accuracy.¹

```
for i := 1 to L do begin
    v := randomly drawn negative instance
    T := T - any term that is satisfied by v
end
return T
```

The algorithm returns the set of terms remaining in *T*, with the interpretation that their disjunction is the concept that was learned by the algorithm. This method simply examines a fixed number of negative instances and removes any term from *T* that would have caused one of the negative instances to be satisfied.²

The following sections describe algorithms for learning action maps in *k*-DNF from reinforcement and present the results of an empirical comparison of their

¹This choice is not relevant to our reinforcement-learning scenario—the details are described in Valiant's papers [Valiant, 1984, Valiant, 1985].

²Valiant's presentation of the algorithm defines *T* to be the set of conjunctive terms of length *k* or less over the set of atoms and their negations; however, because any term of length less than *k* can be represented as a disjunction of terms of length *k*, we use a smaller set *T* for simplicity in exposition and slightly more efficient computation time.

performance. For each algorithm, the inputs are bit-vectors of length *M*, plus a distinguished reinforcement bit; the outputs are single bits.

3 Connectionist Methods for Learning *k*-DNF

There has been interesting work in the connectionist community on learning from reinforcement, which is relevant to our goals because it focuses on using more efficient algorithms to learn action maps in a restricted class of functions. This section will describe three connectionist methods: a linear reinforcement-comparison method, a multi-layer backpropagation method, and a hybrid method that combines Valiant's algorithm for concept learning with the linear reinforcement-comparison method.

These and other algorithms will be described in a standard form consisting of three components: *s*₀ is the initial internal state of the algorithm; *u*(*s*, *i*, *a*, *r*) is the update function, which takes the state of the algorithm *s*, the last input *i*, the last action *a*, and the reinforcement value received *r*, and generates a new algorithm state; and *e*(*s*, *i*) is the evaluation function, which takes an algorithm state *s* and an input *i*, and generates an action.

3.1 Linear Reward-Comparison Method

Most of the connectionist methods are simple single-layer algorithms that can learn action maps in the class of linearly separable functions [Widrow *et al.*, 1973, Sutton, 1984, Barto and Anandan, 1985]. Sutton [Sutton, 1984] performed extensive experiments on such methods and found that *reinforcement-comparison* algorithms tend to have the best performance. The equations below define Algorithm 8 from his dissertation [Sutton, 1984], which uses a version of the Widrow-Hoff or Adaline [Widrow and Hoff, 1960] weight-update algorithm.

The input is represented as an M-dimensional vector i. The internal state, s₀, consists of two M-dimensional vectors, v and w.

$$u(s, i, a, r) = \begin{array}{l} \text{let } p := \sum_{j=1}^M v_j i_j \\ \text{for } j = 1 \text{ to } M \text{ do begin} \\ \quad w_j := w_j + \alpha(r - p)(a - 1/2)i_j \\ \quad v_j := v_j + \beta(r - p)i_j \\ \text{end} \end{array}$$

$$e(s, i) = \begin{cases} 1 & \text{if } \sum_{j=1}^M w_j i_j + \nu > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\alpha > 0$, $0 < \beta < 1$, and ν is a normally distributed random variable of mean 0 and standard deviation δ_y .

The output, *e*(*s*, *i*), has value 1 or 0 depending on the inner product of *w* and *i* and the value of the random variable ν . The addition of the random value causes the algorithm to "experiment" by occasionally performing actions that it would not otherwise have

taken. The updating of the vector w is somewhat complicated: each component is incremented by a value with four terms. The first term, α , is a constant that represents the learning rate. The next term, $r - p$, represents the difference between the actual reinforcement received and the predicted reinforcement, p . This serves to normalize the reinforcement values: the absolute value of the reinforcement signal is not as important as its value relative to the average reinforcement that the agent has been receiving. The predicted reinforcement, p , is generated using a standard linear associator that learns to associate input vectors with reinforcement values by setting the weights in vector v . The third term in the update function for w is $a - 1/2$: it has constant absolute value and the sign is used to encode which action was taken. The final term is i_j , which causes the j th component of the weight vector to be adjusted in proportion to the j th value of the input.

The space required for the state, as well as time for both update and evaluation operations is $O(M)$, where M is the number of input bits.

3.2 Multi-layer Back-propagation Method

Error back-propagation is a method for training connectionist networks that are comprised of multiple layers. Anderson [Anderson, 1986] has designed a connectionist system with multiple layers that uses backpropagation as a method for learning from reinforcement.

Anderson's system uses two networks: one for learning to predict reinforcement and one for learning which action to take. Each of these is a two-layer network, with all of the hidden units connected to all of the inputs and all of the inputs and hidden units connected to the outputs. The system was designed to work in worlds with delayed reinforcement (which are discussed here at greater length in Section 6), but it is easily modified to work in our simpler domain. This algorithm is rather complex, so space does not allow it to be described further. A clear description can be found in Anderson's dissertation [Anderson, 1986].

This method is theoretically able to learn very complex functions, but tends to require many training instances before it converges. The time and space complexity for this algorithm is $O(MH)$, where M is the number of input bits and H is the number of hidden units.

3.3 A Hybrid Algorithm

Given our interest in restricted classes of functions, we can construct a new hybrid algorithm for learning action maps in k -DNF. It hinges on the simple observation that any such function can be expressed as a linear combination of terms in the set T , where T is the set of conjunctive terms of length k over the set of atoms (corresponding to the input bits) and their negations. It is possible to take the original M -bit input signal and transduce it to a wider signal that is the result of evaluating each member of T on the original inputs. We can use this new signal as input to a relatively simple connectionist learning algorithm, such as

the one described in Section 3.1 above.

If there are M input bits, the set T has size $C(2M, k)$ because we are choosing from the set of bits and their negations. However, we can eliminate all elements that contain both an atom and its negation, yielding a set of size $2^k C(M, k)$. The space required by the algorithm, as well as the time to update the internal state or to evaluate an input instance, is proportional to the size of T , and thus, $O(M^k)$. It is important to note that this algorithm (as well as the other three discussed in this paper) is *strictly incremental*: its time and space requirements depend only on the size of the input and on the fixed parameter k and do not increase over the course of a run.

4 Interval-Estimation Algorithm for k -DNF

The interval-estimation algorithm for k -DNF is, like the hybrid algorithm described in Section 3.3, based on Valiant's algorithm, but the interval-estimation algorithm uses standard statistical estimation methods rather than connectionist weight-adjustments. The technique of interval-estimation has also been applied to other reinforcement-learning problems [Kaelbling, forthcoming].

4.1 General Description

This section will describe the algorithm independent of particular statistical tests, which will be introduced in the next section. We shall need the following definitions, however. An input bit-vector *satisfies* a term whenever all the bits mentioned positively in the term have value 1 in the input and all the bits mentioned negatively in the term have value 0 in the input. The quantity $er(t, a)$ is the expected value of the reinforcement that the agent will gain, per trial, if it generates action a whenever term t is satisfied by the input and action $\neg a$ otherwise. The quantity $ubr_a(t, a)$ is the upper bound of a $100(1 - \alpha)\%$ confidence interval on the expected reinforcement gained from performing action a whenever term t is satisfied by the input. We can now give the formal definition of the algorithm:

$s_0 =$ the set T , with a collection of statistics associated with each member of the set

$e(s, i) =$ for each t in S
 if i satisfies t and
 $ubr_a(t, 1) > ubr_a(t, 0)$ and
 $\Pr(er(t, 1) = er(t, 0)) < \beta$
 then return 1
 return 0

$u(s, i, a, r) =$ for each t in S
 update_term_statistics(t, i, a, r)
 return s

At any moment in the operation of this algorithm, we can extract a symbolic description of its current hypothesis. It is the disjunction of all terms t such that $ubr_a(t, 1) > ubr_a(t, 0)$ and $\Pr(er(t, 1) = er(t, 0)) < \beta$. This is the k -DNF expression according to which the agent is choosing its actions.

The evaluation criterion is chosen in such a way as to make the important trade-off between acting to gain information and acting to gain reinforcement. A naive method would be for each term to generate a 1 whenever action 1 has had a higher success rate than action 0. This would be a very bad strategy, however, because if the first trial of action 0 failed, its success rate would be 0, causing action 0 never to be chosen again. The interval estimation method works because of the fact that the value of ubr can be high for two reasons. It may be high because the confidence interval is very large due to the action not having been tried very often—this will cause the action to be chosen in order to gain information. The upper bound may also be high because the confidence interval is small and the action has a genuinely high payoff—this will cause an action to be chosen in order to gain reinforcement. At the beginning of a course of execution of this algorithm, actions are chosen almost at random, until the upper bound of the worse action is driven down by sampling, while the upper bound of the other stays high. The value of α determines the size of the confidence interval: when it is small the confidence interval is large and the algorithm is very conservative. It is not likely to converge to the wrong action, but it may take a long time to converge. As α is increased, the confidence intervals become smaller, the learning rate faster, and the chance of gross error higher.

Let the *equivalence probability* of a term be the probability that the expected reinforcement is the same no matter what choice of action is made when the term is satisfied. The second requirement for a term to cause a 1 to be emitted is that the equivalence probability be small. Without this criterion, terms for which no action is better will, roughly, alternate between choosing action 1 and action 0. Because the output of the entire algorithm will be 1 whenever any term has the value 1, this alternation of values can cause a large number of wrong answers. Thus, if we can convince ourselves that a term is irrelevant by showing that its choice of action makes no difference, we can safely ignore it.

4.2 Statistics

In the simple reinforcement-learning scenario we are considering, the necessary statistical tests are also quite simple. For each term, we store the following statistics: n_0 , the number of trials of action 0; s_0 , the number of successes of action 0; n_1 , the number of trials of action 1; and s_1 , the number of successes of action 1. These statistics are incremented only when the associated term is satisfied by the current input instance.

If n is the number of trials and s the number of successes arising from a series of Bernoulli trials with success probability p , the upper bound of a $100(1 - \alpha)$ percent confidence interval for p can be approximated by

$$h(s, n, \alpha) = \frac{\frac{s}{n} + \frac{z_{\alpha/2}^2}{2n} + \frac{z_{\alpha/2}}{\sqrt{n}} \sqrt{\left(\frac{s}{n}\right) \left(1 - \frac{s}{n}\right) + \frac{z_{\alpha/2}^2}{4n}}}{1 + \frac{z_{\alpha/2}^2}{n}}$$

where $z_{\alpha/2}$ is such that $\Pr(Z \geq z_{\alpha/2}) = \Pr(Z \leq -z_{\alpha/2}) = \alpha/2$ when Z is a standard normal random variable [Larsen and Marx, 1986]. This allows us to define $ubr_{\alpha}(t, 0)$ as $h(s_0, n_0, \alpha)$ and $ubr_{\alpha}(t, 1)$ as $h(s_1, n_1, \alpha)$, where s_0 , n_0 , s_1 , and n_1 are the statistics associated with term t .

To test for equality of the underlying Bernoulli parameters, we use a two-sided test at the β level of significance that rejects the hypothesis that the parameters are equal whenever

$$\frac{\frac{s_0}{n_0} - \frac{s_1}{n_1}}{\sqrt{\left(\frac{s_0 + s_1}{n_0 + n_1}\right) \left(1 - \frac{s_0 + s_1}{n_0 + n_1}\right) \frac{(n_0 + n_1)}{n_0 n_1}}} \text{ is either } \begin{cases} \leq -z_{\beta/2} \\ \text{or} \\ \geq +z_{\beta/2} \end{cases}$$

where $z_{\beta/2}$ is a standard normal deviate [Larsen and Marx, 1986]. Because sample size is important for this test, the algorithm is slightly modified to ensure that, at the beginning of a run, each action is chosen a minimum number of times, referred to by the parameter β_{min} .

The complexity of this algorithm is the same order as that of the hybrid connectionist algorithm of Section 3.3, namely $O(M^k)$.

5 Empirical Comparison

This section reports the results of a set of experiments designed to compare the performance of the algorithms discussed in this paper.

5.1 Algorithms and Environments

The following algorithms were tested in these experiments:

- LINCONN Linear reinforcement-comparison algorithm
- LINCONN+ Linear reinforcement-comparison with an extra input wired to have a constant value
- CONNKDNF Hybrid connectionist algorithm for k -DNF
- IEKDNF Interval-estimation algorithm for k -DNF
- BP Anderson's error back-propagation algorithm
- IE Basic interval-estimation algorithm

The basic interval-estimation algorithm IE [Kaelbling, forthcoming] is included as a yardstick; it is computationally much more complex than the other algorithms and will very likely out-perform them.

Each of the algorithms was tested in three different environments. The environments are called *binomial Boolean expression worlds* and can be characterized by the following parameters: M , $expr$, p_{1s} , p_{1n} , p_{0s} , and p_{0n} . The parameter M is the number of input bits; $expr$ is a Boolean expression over the input bits; p_{1s} is the probability of receiving reinforcement value 1 given that action 1 is taken when the input instance satisfies $expr$; p_{1n} is the probability of receiving reinforcement value 1 given that action 1 is taken when the input instance does not satisfy $expr$; p_{0s} is the probability of receiving reinforcement value 1 given that action 0

Task	M	p_{1s}	p_{1n}	p_{0s}	p_{0n}
1	3	.6	.4	.4	.6
2	3	.9	.1	.1	.9
3	6	.9	.5	.6	.8

Table 1: Parameters of test environments for k -DNF experiments.

is taken when the input instance satisfies $expr$; and p_{0n} is the probability of receiving reinforcement value 1 given that action 0 is taken when the input instance does not satisfy $expr$. Input vectors are chosen by the world according to a uniform probability distribution.

Table 1 shows the values of these parameters for each task. The first task has the simple, linearly separable expression $(i_0 \wedge i_1) \vee (i_1 \wedge i_2)$; what makes it difficult is the small separation between the reinforcement probabilities. Task 2 has highly differentiated reinforcement probabilities, but the function to be learned, $(i_0 \wedge \neg i_1) \vee (i_1 \wedge \neg i_2) \vee (i_2 \wedge \neg i_0)$, is a complex exclusive-or. Finally, Task 3 is the simple conjunctive function, $i_2 \wedge \neg i_5$, but all of the reinforcement probabilities are high and there are 6 input bits rather than only 3.

5.2 Parameter Tuning

Each of the algorithms has a set of parameters. For both IEKDNF and CONNKDNF, $k = 2$. The simple connectionist algorithms LINCONN and LINCONN+ as well as CONNKDNF have parameters α , β , and σ . Following Sutton [Sutton, 1984], parameters β and σ in CONNKDNF, LINCONN, and LINCONN+ will be fixed to have values .1 and .3, respectively. The IEKDNF algorithm has two confidence-interval parameters, $z_{\alpha/2}$ and $z_{\beta/2}$, and a minimum age for the equality test β_{min} , while the IE algorithm has only $z_{\alpha/2}$. Finally, the BP algorithm has a large set of parameters: β , learning rate of the evaluation output units; β_h , learning rate of the evaluation hidden units; ρ , learning rate of the action output units; and ρ_h , learning rate of the action hidden units. In each of the tasks, the BP algorithm had as many hidden units as inputs.

All of the parameters for each algorithm were be chosen to optimize the behavior of that algorithm on the chosen task. The success of an algorithm was measured by the average reinforcement received per tick, averaged over the entire run. For each algorithm and environment, a series of 100 trials of length 3000 were run with different parameter values. Table 2 shows the best set of parameter values found for each algorithm-environment pair.

5.3 Results

Having chosen the best parameter values for each algorithm and environment, the performance of the algorithms was compared on runs of length 3000 using the parameter settings of Table 2. The performance metric was average reinforcement per tick, averaged over the entire run. The results are shown in Table 3, together with the expected reinforcement of executing a completely random behavior (choosing actions

ALG-TASK	1	2	3
LINCONN			
α	.0625	.125	.125
LINCONN+			
α	.125	.0625	.25
CONNKDNF			
α	.125	.25	.03125
IEKDNF			
$z_{\alpha/2}$	3	3.5	2.5
$z_{\beta/2}$	1	2.5	3.5
β_{min}	15	5	25
BP			
β	.1	.25	.1
β_h	.2	.3	.05
ρ	.15	.15	.35
ρ_h	.2	.05	.1
IE			
$z_{\alpha/2}$	3.0	1.5	2.5

Table 2: Best parameter values for each k -DNF algorithm in each environment.

ALG-TASK	1	2	3
LINCONN	.5329	.7418	.7769
LINCONN+	.5456	.7459	.7722
CONNKDNF	.5783	.8903	.7825
IEKDNF	.5789	.8900	.7993
BP	.5456	.7406	.7852
IE	.5827	.8966	.7872
random	.5000	.5000	.6750
optimal	.6000	.9000	.8250

Table 3: Average reinforcement for k -DNF problems over 100 runs of length 3000

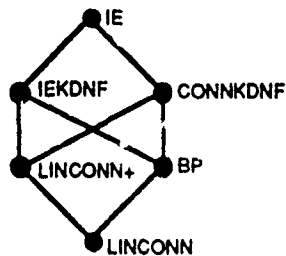


Figure 1: Significant dominance partial order among k -DNF algorithms for Task 1.

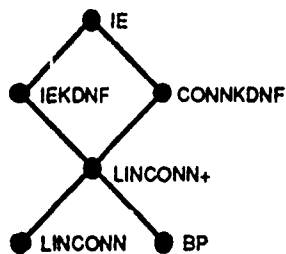


Figure 2: Significant dominance partial order among k -DNF algorithms for Task 2.

0 and 1 with equal probability) and of executing the optimal behavior. These results do not tell the entire story, however. It is important to test for statistical significance to be relatively sure that the ordering of one algorithm over another did not arise by chance. Figures 1, 2 and 3 show, for each task, a pictorial representation of the results of a 1-sided t -test applied to each pair of experimental results. The graphs encode a partial order of significant dominance, with solid lines representing significance at the .95 level and dashed lines representing significance at the .85 level.

With the best parameter values for each algorithm, it is also of some interest to compare the rate at which performance improves as a function of the number of training instances. Figures 4, 5, and 6 show superimposed plots of the learning curves for each of the algorithms. Each point represents the average reinforcement received over a sequence of 100 steps, averaged over 100 runs of length 3000.

5.4 Discussion

On Tasks 1 and 2 the basic interval-estimation algorithm, IE, performed significantly better than any of the other algorithms. The magnitude of its superiority, however, is not extremely great—Figures 4 and 5 reveal that the IEKDNF and CONNKDNF algorithms have similar performance characteristics both to each other and to IE. On these two tasks, the overall performance of IEKDNF and CONNKDNF were not found to be significantly different.

The backpropagation algorithm, BP, performed considerably worse than expected on Tasks 1 and 2. It is very difficult to tune the parameters for this algorithm, so its bad performance may be explained by

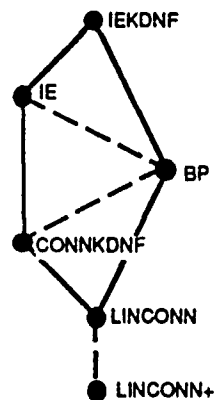


Figure 3: Significant dominance partial order among k -DNF algorithms for Task 3.

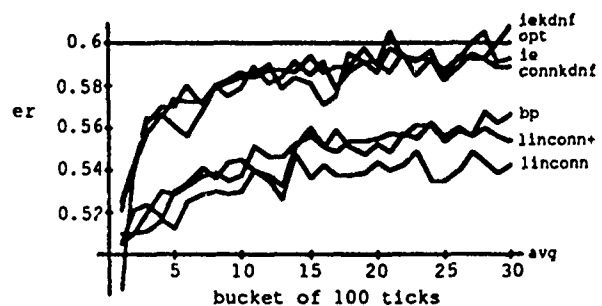


Figure 4: Learning curves for Task 1.

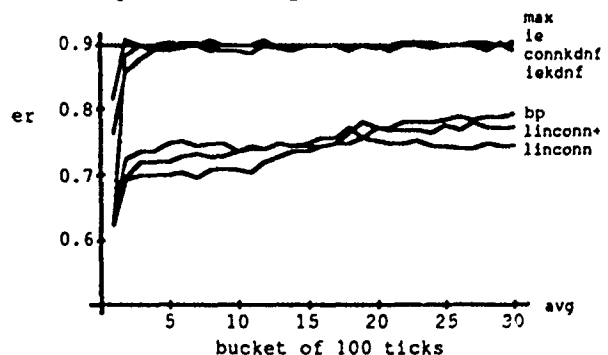


Figure 5: Learning curves for Task 2.

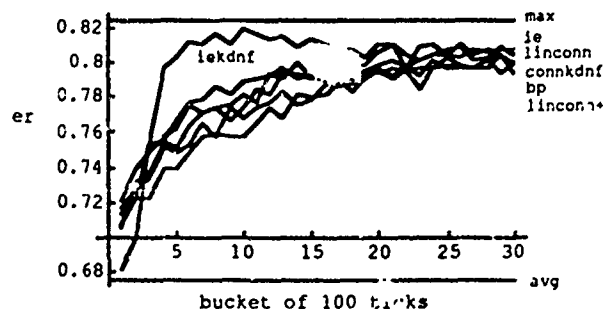


Figure 6: Learning curves for Task 3.

a sub-optimal setting of parameters.³ However, it is possible to see in the learning curves of Figures 4 and 5 that the performance of BP was still increasing at the ends of the runs. This may indicate that with more training instances it would eventually converge to optimal performance.

The simple linear connectionist algorithms performed poorly on both Tasks 1 and 2. This poor performance was expected on Task 2, because such algorithms are known to be unable to learn non-linearly-separable functions. Task 1 is difficult for these algorithms because, during the execution of the algorithm, the evaluation function is often too complex to be learned by the simple linear associator. Adding a constant input value to the simple linear connectionist algorithm made a significant improvement in performance; this is not surprising, because it allows discrimination hyperplanes that do not pass through the origin of the space to be found.

Task 3 reveals many interesting strengths and weaknesses of the algorithms. One of the most interesting is that IE is no longer the best performer. Because the target function is simple and there is a larger number of input bits, the ability to generalize across input instances becomes important. The IEKDNF algorithm is able to find the correct hypothesis early during the run (this is apparent in the learning curve of Figure 6). However, because the reinforcement values are not highly differentiated and because the size of the set T is quite large, it begins to include extraneous terms due to statistical fluctuations in the environment, causing slightly degraded performance.

The IE, BP, and CONNKDNF algorithms all have very similar performance on Task 3, with the linear connectionist algorithms performing slightly worse, but still reasonably well.

6 Relaxing the Assumptions

This section will discuss the consequences of relaxing the assumptions made at the beginning of this paper, especially in the context of the two better-performing algorithms, IEKDNF and CONNKDNF. In some cases, simple changes can be made to the algorithms that will allow them to work in the more general situations. In others, there are theoretical problems that make extensions difficult. Each of the concrete extensions proposed to the IEKDNF algorithm has been implemented and tested.

Thus far we have assumed that the agent has only two possible actions. Many of the early learning-automata algorithms are directly applicable to problems with more than two actions. It has also been shown [Kaelbling, forthcoming] that the problem of generating actions specified by N output bits can be

solved by N interconnected modules that learn to generate one output bit from reinforcement. Thus, the algorithms presented here could be applied, using this method, to problems with many possible outputs.

The problem of delayed reinforcement has been addressed by Sutton [Sutton, 1988] and Watkins [Watkins, 1989], among others. Sutton's solution, called the *temporal difference method* (TD) can be abstracted away from the particular reinforcement-learning mechanism being used. It provides a module that learns to transduce the delayed reinforcement signal that is coming from the world into an immediate reinforcement signal that evaluates each state of the world to be the expected future reward based on the agent's current strategy. Because this local reinforcement signal must be learned, using a TD module violates a different one of our assumptions: that the expected reinforcement of performing an action in a situation be fixed over the course of a run. This will be addressed below.

If the reinforcement provided by the world cannot be modeled as independent trials of some sort, then it is very difficult to use explicit statistical methods. The connectionist algorithms are implicitly statistical and would also have trouble in such worlds. However, if the trials are independent, we have a variety of different statistical models available. The CONNKDNF algorithm, as presented, can be used when the reinforcement is real-valued. The IEKDNF algorithm can be implemented with different statistical tests. For instance, if we know that the reinforcement values for each input-action pair are normally distributed, we can use standard statistical methods to construct confidence intervals and to test for equality of means. If we have no model, we can use non-parametric methods.

Finally, we consider the case of having the expected reinforcement of performing an action in a situation change during the course of a run. The CONNKDNF algorithm will work in such cases, although it might be necessary to adjust its parameters. The statistically-based IEKDNF algorithm can be modified to work, by causing its statistics to decay over time. If an action has not been tried for a long time, its n value will slowly decay, which will cause its confidence interval to grow larger. Eventually it will grow large enough for that action to be chosen again. If the action has good results, the policy will be changed to favor this action.

7 Conclusion

From this study, we can see that it is useful to design algorithms that are tailored to learning certain restricted classes of functions. The two specially-designed algorithms far out-performed standard methods of comparable complexity. The CONNKDNF and IEKDNF algorithms each have their strengths and weaknesses. It is possible that CONNKDNF may out-perform IEKDNF to some extent because in CONNKDNF each term gets to contribute to the answer with different degrees. This avoids errors that occur in IEKDNF when a single term is barely over the threshold for gen-

³In the parameter tuning phase, the parameters were varied independently—it may well be necessary to perform gradient-ascent search in the parameter space, but that is a computationally difficult task, especially when the evaluation of any point in parameter space may have a high degree of noise.

erating a 1. On the other hand, the state of IEKDNF has internal semantics that are clear and directly interpretable in the language of classical statistics. This simplifies the process of extending the algorithm to apply to other types of worlds in a principled manner.

Important future work will be to identify other restricted classes of functions that can be learned efficiently and effectively from reinforcement and demonstrate that these classes contain functions that solve interesting and important problems from the real world.

Acknowledgments

Thanks to Stan Rosenschein for providing financial and moral support and to Rich Sutton for helpful discussions of connectionist and statistical reinforcement learning methods.

References

- [Anderson, 1986] Charles W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Amherst, Massachusetts, 1986.
- [Barto and Anandan, 1985] A. G. Barto and P. Anandan. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360-374, 1985.
- [Berry and Fristedt, 1985] Donald A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London, 1985.
- [Enderton, 1972] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, New York, 1972.
- [Kaelbling, 1989a] Leslie Pack Kaelbling. A formal framework for learning in embedded systems. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 350-353, Ithaca, New York, 1989. Morgan Kaufmann.
- [Kaelbling, 1989b] Leslie Pack Kaelbling. Foundations of learning in autonomous agents. In *Proceedings of the Workshop on Representation and Learning Autonomous Agents*, Lagos, Portugal, 1989.
- [Kaelbling, forthcoming] Leslie Pack Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, Stanford, California, forthcoming.
- [Larsen and Marx, 1986] Richard J. Larsen and Morris L. Marx. *An Introduction to Mathematical Statistics and its Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [Narendra and Thathachar, 1989] Kumpati Narendra and M. A. L. Thathachar. *Learning Automata. An Introduction*. Prentice-Hall, Englewood, New Jersey, 1989.
- [Sutton, 1984] Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, Massachusetts, 1984.
- [Sutton, 1988] Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9-44, 1988.
- [Valiant, 1984] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134-1142, 1984.
- [Valiant, 1985] L. G. Valiant. Learning disjunctions of conjunctions. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 560-566, Los Angeles, California, 1985. Morgan Kaufmann.
- [Watkins, 1989] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, 1989.
- [Widrow and Hoff, 1960] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *IRE WESCON Convention Record*, New York, New York, 1960. Reprinted in *Neurocomputing: Foundations of Research*, edited by James A. Anderson and Edward Rosenfeld, MIT Press, Cambridge, Massachusetts, 1988.
- [Widrow et al., 1973] Bernard Widrow, Narendra K. Gupta, and Sidhartha Maitra. Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(5):455-465, 1973.
- [Williams, 1986] Ronald J. Williams. Reinforcement learning in connectionist networks: A mathematical analysis. Technical report, Institute for Cognitive Science, University of California, San Diego, La Jolla, California, 1986.

E Intermediate Vision: Architecture, Implementation, and Use

Intermediate vision: Architecture, implementation, and use

David Chapman

Technical Report No. TR-90-06

October, 1990

Abstract

This paper describes an implemented architecture for intermediate vision. By integrating a variety of intermediate visual mechanisms and putting them to use in support of concrete activity, the implementation demonstrates their utility. The system, SIVS, models psychophysical discoveries about visual attention and search. It is designed to be efficiently implementable in biologically realistic hardware.

SIVS addresses five fundamental problems. *Visual attention* is required to restrict processing to task-relevant locations in the image. *Visual search* finds such locations. *Visual routines* are a means for nonuniform processing based on task demands. *Intermediate objects* keep track of intermediate results of this processing. *Visual operators* are a set of relatively abstract, general-purpose primitives for spatial analysis, out of which visual routines are assembled.

This report describes research done in part at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-85-K-0124. This research was also done in part at Teleos Research and supported by the Air Force Office of Scientific Research under contract F49620-89-C-0055, by the Defense Advanced Research Projects Agency under NASA contract NAS2-13229, and by Teleos Internal Research and Development funds.

Copyright © David Chapman, 1990. All rights reserved.

Contents

1	Introduction	2
2	Visual attention	5
3	Visual search	9
3.1	Psychophysics of visual search	9
3.2	An architecture for visual search	10
3.3	Extensions	14
3.4	Related computational work	16
3.5	Open questions	17
4	Visual routines	17
5	Intermediate objects	21
6	Visual operators	24
6.1	Criteria on visual operators	24
6.2	SIVS's visual operators	25
6.2.1	Visual attention and search	25
6.2.2	Tracking	26
6.2.3	Distances, directions, and angles	27
6.2.4	Marker, line, and ray manipulation	28
6.2.5	Activation	28
6.2.6	Blanking	29
7	Visually guided activity	29
8	Conclusions	44
8.1	Outstanding problems	44
8.2	Evaluating the visual operators	45
8.3	Successes	46

1 Introduction

This paper presents an implemented architecture for *intermediate vision*: the mechanisms that connect bottom-up early vision with later, task-specific processing. The system, SIVS,

- models relevant psychophysical results,
- obeys the constraints imposed by biologically plausible hardware,
- addresses key computational problems in vision that are often passed over, and
- integrates a variety of mechanisms to support complex activity in a realistic task domain.

Background and summary

Unlike some machine vision systems which seek engineering solutions by whatever means, SIVS is intended to model specifically *biological* vision. The inputs and the first few *early* stages of mammalian visual processing are relatively well understood as a result of neurophysiological studies and computational modeling [20, 30, 31]. We know less about the nature of processing after early vision and before the outputs. Computational studies have mainly addressed the problem of object recognition by shape matching. Object recognition, often referred to as *late* vision, is an important part of visual processing, but there is much evidence (reviewed later in the paper) for *intermediate* visual processes in addition. Vision does not leap from early representations to final outputs in a single step. Unfortunately, relatively little is known about intermediate visual processing. There is little relevant neurophysiological evidence, for example.

Progress at this point seems to require the construction of plausible models which can suggest questions for neuroscientific, psychological, and computational experiments. Such a model must respect the evidence that is available, even if it is scanty; the model this paper proposes is informed by psychophysical, neurocomputational, and engineering evidence. SIVS models the psychophysics of visual attention and search in detail. It is designed to be implementable in slow, massively parallel, locally connected hardware, such as that found in the brain. It is based on an engineering analysis of the intermediate vision task, it has proven adequate to support visually-guided activity in a complex domain.

The intermediate visual processes I posit perform non-local computations with representations of portions of the image. Thus they contrast with early vision, which is concerned with local computations, and with late vision, which produces representations of external objects. They also span the gap between early and late processing in terms of the sorts of encodings of information used. Early vision maps the retinal inputs point-by-point into *retinotopic* representations. Late vision probably encodes its outputs with what I will call *compact encodings*, small groups of neurons which together represent a particular property of a scene. These properties might be coded as boolean values or continuous scalars. For example Perrett *et al.* describe experiments that suggest that individual monkey neurons respond selectively to faces [39]. (The interpretation of these and related experiments is still controversial; see [31] for a review.) The input encodings for late vision are unknown, and in any case not well defined since the scope of "late" vision is itself not well defined. However, it seems likely that the inputs are also in the form of compact encodings of properties of regions of the image. Thus, the intermediate visual

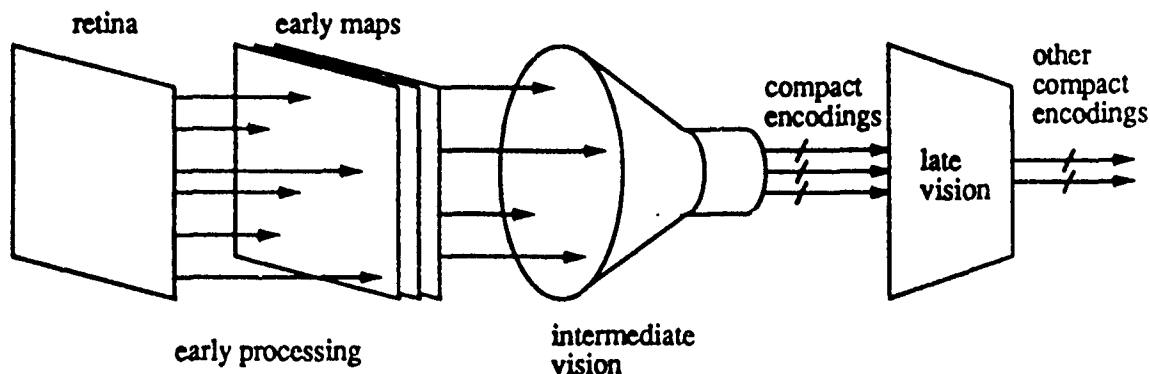


Figure 1: Early, intermediate, and late vision. Early processing computes, point by point, retinotopic maps from the retinal image; intermediate vision reduces these maps to compact encodings; late vision computes exclusively with compact encodings.

processes I propose fill the gap by reducing retinotopic representations to compact ones (figure 1).

These intermediate processes are intended to be roughly equidistant from early vision and final outputs. Thus, should the proposed mechanisms be found to model human performance, they will place considerable constraint on the remaining parts of the puzzle. In order to exploit this constraint, it would be necessary to interface SIVS with realistic models of early and late processing. I haven't done this; I chose SIVS's domain so that, although it was of practical use in a broader research program, I did not have to implement early vision or general object recognition. However, this paper specifies the interfaces between the intermediate processes I implemented and early and late vision; section 8 discusses some remaining difficulties.

This paper addresses five issues that arise in the computation of nonlocal properties of images. These issues are sufficiently fundamental that it seems that any intermediate vision system will have to address them; they are relatively unstudied in the computational vision literature, however.

- Visual processing must be applied selectively to task-relevant regions of the image.
- The visual system must therefore be able to find regions of the image with task-relevant properties.
- Visual processing must be serial in part, with various operations performed in sequence and according to environmental conditions.
- Thus, the system must be able to keep track of intermediate results of visual computations.
- The enormous variety of visual tasks suggests that visual processing must allow the development of new patterns of processing in response to new task requirements.

SIVS addresses these issues with

- *visual attention*, which restricts access to image properties to a "spotlight" of attention;
- *visual search*, which can direct the attentional spotlight to regions of the image satisfying particular criteria;
- *visual routines*, task- and situation-specific sequential patterns of visual processing;
- *intermediate objects*, image-centered representations for intermediate results of visual routines; and
- *visual operators*, a set of general-purpose, relatively abstract primitives, which are combined to form visual routines.

These mechanisms have been proposed by others on psychophysical and speculative computational grounds. However, many aspects of these proposed mechanisms have previously been left vague. As has often been the case in cognitive science, a computer implementation forced complete specification, thereby uncovering a variety of new issues. Engineering considerations led to the development of new mechanisms which may or may not be found in human vision. These issues and proposed mechanisms may now be subjected to psychophysical scrutiny. Further, mechanisms such as visual search have typically been studied in isolation. SIVS demonstrates that it is possible to integrate several such mechanisms to achieve synergistic power.

Studies linking perception and action have been rare in artificial intelligence. An exception has been work in robotics, but the vision systems used there have tended to be ad-hoc and not psychologically motivated. SIVS is designed to support visually-guided activity in a psychologically realistic way. This is important because the psychophysical studies on which the proposed mechanisms are based were conducted on isolated, highly specific, artificial tasks. It was, therefore, possible that these mechanisms would have no actual use in a broader context, or that they would have to be used very differently under ecologically valid circumstances. That SIVS is able to support complex activity in a realistic domain demonstrates for the first time that these mechanisms are of practical value.

SIVS is part of a larger system called Sonja [7].¹ Sonja integrates advances in vision, natural language pragmatics, and action. Sonja plays a video game called Amazon modeled after a commercial arcade game. Its access to the game world is only via SIVS and via the game's primitive actions.

Outline

Section 2 describes visual attention, the ability to access subsets of the early retinotopic representations. Neurophysiological and psychophysical evidence suggest that visual attention is implemented with a mechanism that routes information from dynamically selected locations

¹The version of SIVS reported on here is improved in several respects over that of [7].

to a central node; this device is a key locus of the reduction of retinotopic representations to compact encodings.

Section 3 describes visual search, the ability to find locations in an image that have specified properties. Visual search has been shown psychophysically to depend on visual attention; in many cases it proceeds by serially enumerating and testing locations. SIVS is the first implemented system that models the psychophysically demonstrated properties of human visual search.

Section 4 describes visual routines, patterns of applications of particular visual operations over time. Visual routines can do geometrical work, such as finding the smallest or leftmost item in a collection, and topological work, such as determining connectedness or containment. The notion of visual routines was proposed by Ullman [60], and SIVS draws heavily on his work. His proposal is sketchy in many respects, however; this paper extends it and renders it specific. Although other researchers have worked in the visual routines framework [28, 45], no one has previously produced an implementation complete enough for application.

Section 5 describes intermediate objects, which are used to keep track of the intermediate results of visual routines as they proceed. There are four sorts of intermediate objects, called markers, lines, rays, and activation planes.

Section 6 describes visual operators, hypothesized bits of brain hardware which perform particular sorts of visual work. Visual routines are sequences of activations of visual operators; the theory hypothesizes that there is a small, innate set of operators. Three typical visual operators find the distance between two points, track a moving object, and find the extent of a homogeneous image region. Section 6 describes the specific set of visual operators used in SIVS and the criteria for choosing them.

Section 7 describes the use of SIVS in guiding activity. Practical use demonstrates that SIVS is adequate to support complex activity in a realistic domain. Visually guided activity is, further, an interesting problem in its own right; and its requirements differ from those of object recognition and other well-studied late visual tasks.

Section 8 presents conclusions, evaluating SIVS and describing successes and outstanding problems.

2 Visual attention

Visual attention is the ability to differentially apply visual processing to a subset of a scene. It is taken as consisting of two components: *overt* visual attention, or gaze direction, which can be observed with an eye tracker; and *covert* visual attention, which is neurally mediated and so can only be observed indirectly. This paper is concerned only with covert attention; [7] briefly describes how overt attention might be incorporated into the model. Following standard usage, I will use "visual attention" to mean covert visual attention when this will not result in confusion.

There are large psychophysical and neuroscientific literatures on visual attention; I will review some of this literature in this section. While the data are uncertain and sometimes contradictory, there is broad agreement about some general facts.

The primary evidence for covert visual attention comes from psychophysical studies, for instance those of Posner *et al.* [40]. In a typical experiment, subjects are required to react to an event such as a light coming on somewhere in the visual field. The results of such experiments are that

- Reaction times are lower when the subjects are told where in the field the event will occur, suggesting that visual resources can more effectively be brought to the detection task when the location of the event is known.
- Covert visual attention is independent of (overt) gaze direction: it operates even when the subjects do not foveate the indicated location [36, 40], and brain lesions that eliminate voluntary eye movements do not affect covert attention [41].²
- Visual attention is at least partly cognitively penetrable and under voluntary control; the event location can be indicated by non-natural cues [40].
- The bulk of the evidence suggests that attention can be directed only to a single contiguous subset of the image [13, 40].³
- The diameter of the attended subset can be varied voluntarily [13, 22, 53, 56]. The possible shapes it can assume and the distinctness of its margin are uncertain [53].⁴

These observations are summarized as the *spotlight* model of attention, in which attention "illuminates" a chosen subset of the image. The precise nature of these subsets is unclear, so I will refer to them neutrally as "locations"; I will discuss them further in section 3.5.

Covert attention interacts closely with early vision. Therefore I will summarize necessary background concerning early vision before proceeding. The retina is a two-dimensional array of light sensors. Neuroscientific study shows that the parts of the brain to which it is immediately connected preserve this retinal topology [31]. This *retinotopic* neural processing comprises early vision. Early vision is coming to be quite well understood [20, 30, 31]. In addition to being retinotopic, early vision is bottom-up and applies uniformly and in parallel over the image. *Bottom-up* visual processing is that which depends only on the retinal image. A process is bottom-up if and only if the same computation occurs whenever the same image is presented. Bottom-up processing, thus, cannot depend on any non-visual contextual factors, on memories or other state, or on the agent's intentions. Early vision produces "unarticulated" output representations, typically a single continuously variable or boolean value at each point in the image. Early visual processing is performed by a set of fixed, innate, retinotopically organized machines called *early maps*. The identity and function of some of these maps are known; new maps are still being discovered and the functional properties of some remain to be determined.

²There may be weak interactions between covert and overt attention. Kröe, for instance, presents evidence that the detectability of a "T" in a background of "L"s is a function of retinal eccentricity [25]. Other researchers (such as Nakayama and Mackeben [36, p. 1639]) have failed to find such effects.

³Earlier studies by Shaw and Shaw [48, 49] suggesting that attention can be split over arbitrary subsets of the image have not been confirmed by more recent work; but see Driver and Baylis [10].

⁴Eriksen and St. James [13] present evidence for an indistinct margin, with processing efficiency decreasing gradually from the center. Farah's results [14] suggest that attention can be directed to oddly-shaped regions, but this may instead be the result of an unrelated activation operation (see section 4).

There appear to be roughly fifteen maps; among them are ones that compute color, edge orientation, stereoscopic depth, and various properties of motion such as speed, direction, and size change.⁵

Koch and Ullman [24] have proposed an *addressing pyramid* as the hardware supporting the attentional spotlight. (This proposal was inspired by neuroanatomical speculations of Crick [8]; similar proposals have been made by Anderson and Van Essen [2], Treisman and Gormican [55], Tsotsos [58], and others.) The addressing pyramid is similar in function to the addressing hardware of a conventional serial computer: it routes information from a selected part of a peripheral array to a central location. In the case of a conventional computer, this information is the contents of a memory location; in the case of the attentional hardware, it is the contents of the early representations in the attended location in the retinotopic array. The pyramid gets its name from its two-dimensional hierarchical tree organization. It consists of a series of exponentially smaller stacked layers that route information upwards to a central node (figure 2). Each level is composed of an array of nodes, each of which selects one of the nodes beneath it to route to its superior. Thus the system as a whole acts as a recursive *winner-take-all network* [17], eventually routing the contents of just one leaf node up to the root. These leaf nodes actually each contain the values of the early representations at one retinotopic location. I will describe how pyramid nodes choose among their subnodes in section 3.

A spotlight of variable diameter can be implemented by having some of the nodes send up a combination of the values of their inferiors, rather than choosing a single one. This has been suggested by Treisman and Gormican [55], who propose that interior nodes in the pyramid can, selectively, pass up the average of the early values of their inferiors, rather than passing up the exact value of a single chosen inferior.

This addressing pyramid, then, is a key locus of the "collapse" of retinotopic representations into compact encodings that is critical of intermediate vision. Koch and Ullman propose implementing the pyramid in terms of a circuit of neuron-like elements; SIVS follows this proposal closely. I will describe the implementation further in section 3.

⁵Early work by Zeki [63, 64] suggested a one-to-one correspondence between retinotopic maps and types of visual information. It is now known that the correspondence is many-to-many [31]. I will ignore this observation for simplicity.

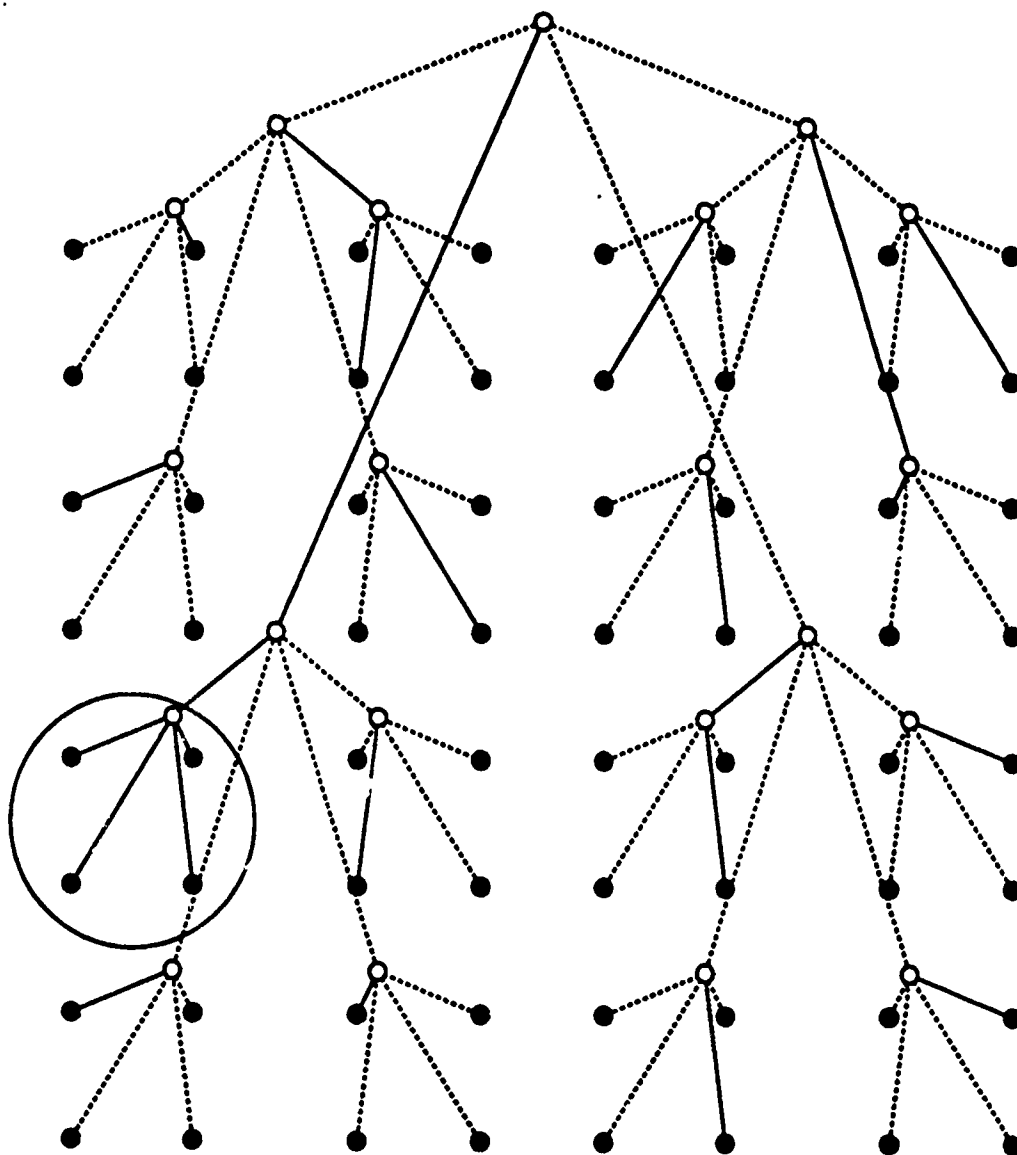


Figure 2: The addressing pyramid. Leaf nodes (solid circles) contain buses compactly encoding early properties. Interior nodes (open circles) pass information from their inferiors up to their superior. Here the encircled region (containing four leaf nodes) is addressed. The interior node immediately above this region passes up the average of the four leaf values, rather than selecting one. The other interior nodes select just one of their inferiors to pass up early properties from. Selected communication paths are drawn as solid lines, deselected paths as dashed ones.

3 Visual search

Visual search is the process of finding locations in the image which have specified properties. Visual search has been extensively studied psychophysically; for surveys, see Julesz [22] and Treisman and Gelade [54]. In most psychophysical experiments, the sorts of properties searched for are very simple: "is red," for example, rather than "is a chair of some sort." Restricting attention to such simple properties has made it possible to isolate the mechanisms that probably underlie more complicated sorts of search. Fortunately, in Amazon (the videogame domain SIVS has been applied to), these simple properties are sufficient to locate the objects that are relevant to any task. Thus it was possible in SIVS to implement the psychophysically demonstrated mechanisms without much speculative extension.

In Sonja visual search is a means to an end, as well as an object of study in itself. Psychophysicists have principally studied visual search in isolation and under artificial conditions. This has begged questions about the interface between these mechanisms and other visual and non-visual processes. For example, questions about the interaction between visual search and segmentation that must be answered to fully specify an implementation have gone unasked. (I'll take this point up in section 3.5.) More seriously, the role of visual search in broader activity has not been addressed. SIVS integrates the visual search mechanisms discovered psychophysically with other visual processes, and (as we'll see in section 7) Sonja further integrates all these visual processes with action to achieve concrete ends.

This section first explains the psychophysical properties of visual search and the brain architecture they suggest, then explains SIVS's implementation of that architecture and compares it with related computational implementations.

3.1 Psychophysics of visual search

The central result of the visual search literature, due to Treisman and her colleagues [54, 55], concerns the distinction between *parallel* and *serial self-terminating* search. The experimental paradigm motivating this distinction examines the time required to determine whether or not an object with specified properties exists somewhere in an artificial scene. The results depend on the nature of the property and also on what objects are found in the scene (figure 3). Tasks varying on these dimensions segregate strongly into two classes. In the first class, the time required is independent of what is in the scene. In the second class, time is a linear function of the number of "distractor" objects in the scene, and on average is twice as long in cases in which the object to be found is not present than when it is present. Treisman interprets the first class as indicating that certain properties are computed in parallel and in constant time over the entire visual field. In cases in which the desired property is one of those computed this way, determining whether or not any object with the property is present can be computed in constant time as a global OR over the resulting retinotopic map. The object, if present, is said to "pop out" of the display, and such properties are called *pop out* properties. Treisman interprets the second class as indicating that, in cases where properties are not computed in parallel, visual attention must be applied sequentially to each location in the field to determine whether or not it has the desired property. In these cases if a single object of the desired type

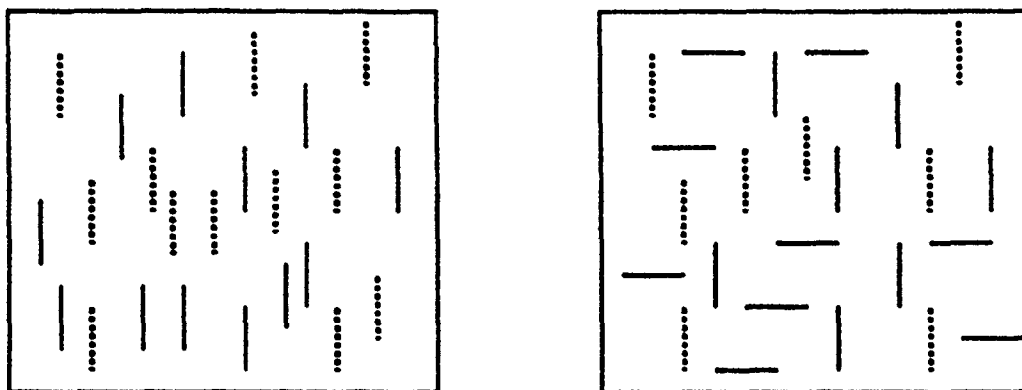


Figure 3: Psychophysical displays requiring, respectively, parallel and serial search. Determining whether or not there is a horizontal line or whether or not there is a dashed line in a display such as the first one takes time independent of the number of objects. Determining whether there is a horizontal dashed line among vertical dashed and horizontal solid lines (as in the second display) requires serial search and takes time linear in the number of objects.

is present, on average half the objects in the field will be examined before it is found; if one is *not* present, every object in the field must be examined. This "serial self-terminating search" accounts neatly for the reaction time data.

Given this paradigm, we can ask what features pop out. Treisman and Gormican [55] report that colors, grey level, line curvature, line orientation, line length, line ends, directions of motion, stereoscopic depth differences, and the proximity and numerosity of clusters of lines are pop out properties. These results are particularly interesting because there is convergent neurophysiological evidence for early retinotopic representations of many of these properties [30, 31]. On the other hand, intersection, line juncture, angle, connectedness, containment, and aspect ratio are not pop out properties. Neither are conjunctions of pop out properties.

Treisman's results have been replicated by many other researchers. Recently, some conflicting data and alternative explanations have been put forth [35, 57, 62]. I have adopted Treisman's model as it is the most generally accepted; new empirical results may force modifications.⁶

3.2 An architecture for visual search

These psychophysical results suggest an architecture like that of figure 4. Early modalities compute retinotopic maps bottom up.⁷ Let us say that an early property consists of a *dimension*

⁶For example, the results of Wolfe *et al.* [62] suggest that the activation maps described in the next section should be continuously graded, rather than binary.

⁷This is an abstraction from neuroscientific results, which show that retinotopic maps are actually computed in a cascade of stages [31]. The work of Moran and Desimone [32] suggests further that these stages are probably interwoven with the attentional pyramid: they found increasing effects of visual attention on the receptive fields

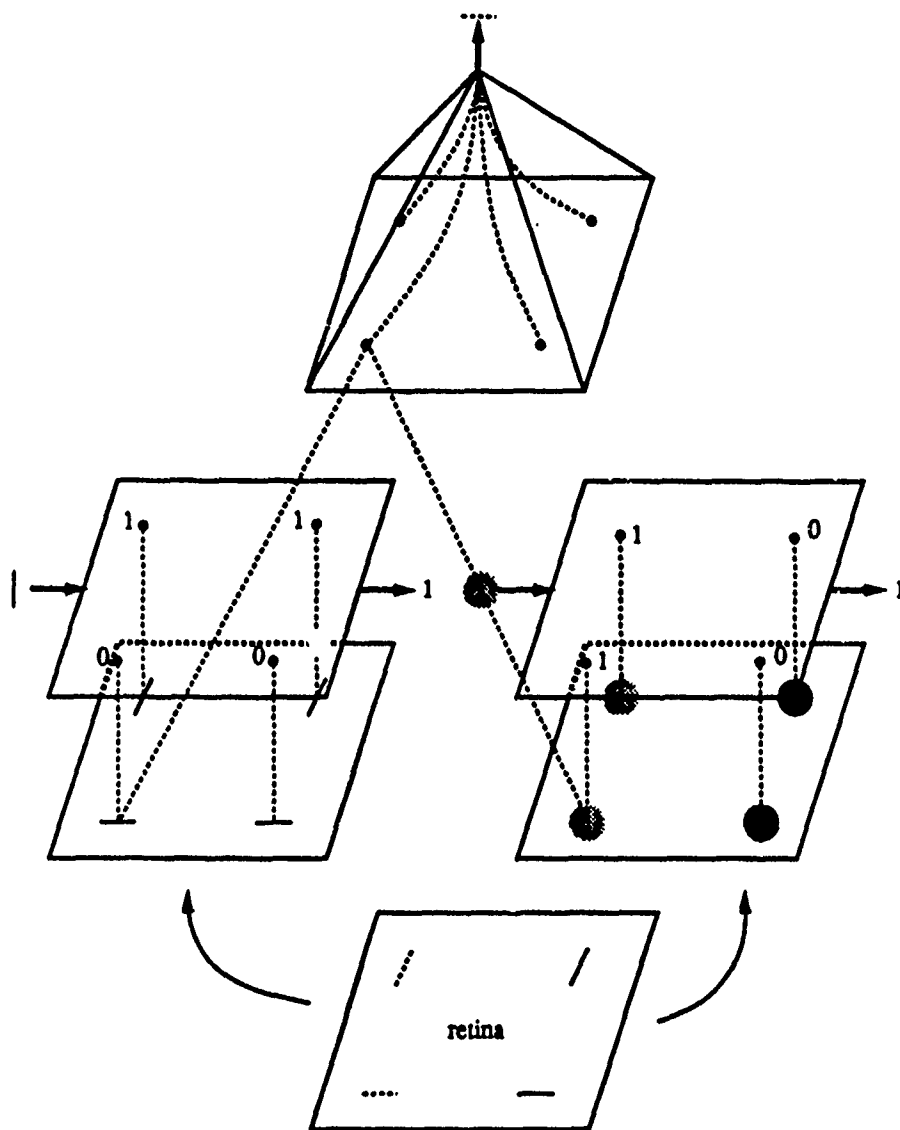


Figure 4: An architecture for visual search. In this example the retina is presented with lines varying in orientation (horizontal, vertical) and color (symbolized by solid and dashed). Two early maps compute these properties. Activation maps compute whether a desired value (vertical or dashed, input from the left) is present in the corresponding early maps at each point. A global OR (output on the right) supports parallel search. The addressing pyramid supports serial search, routing to the root (and thereby combining) all the early properties corresponding to a particular addressed location (the lower left in this case). I have omitted most lines connecting to pyramid leaf nodes to reduce visual clutter.

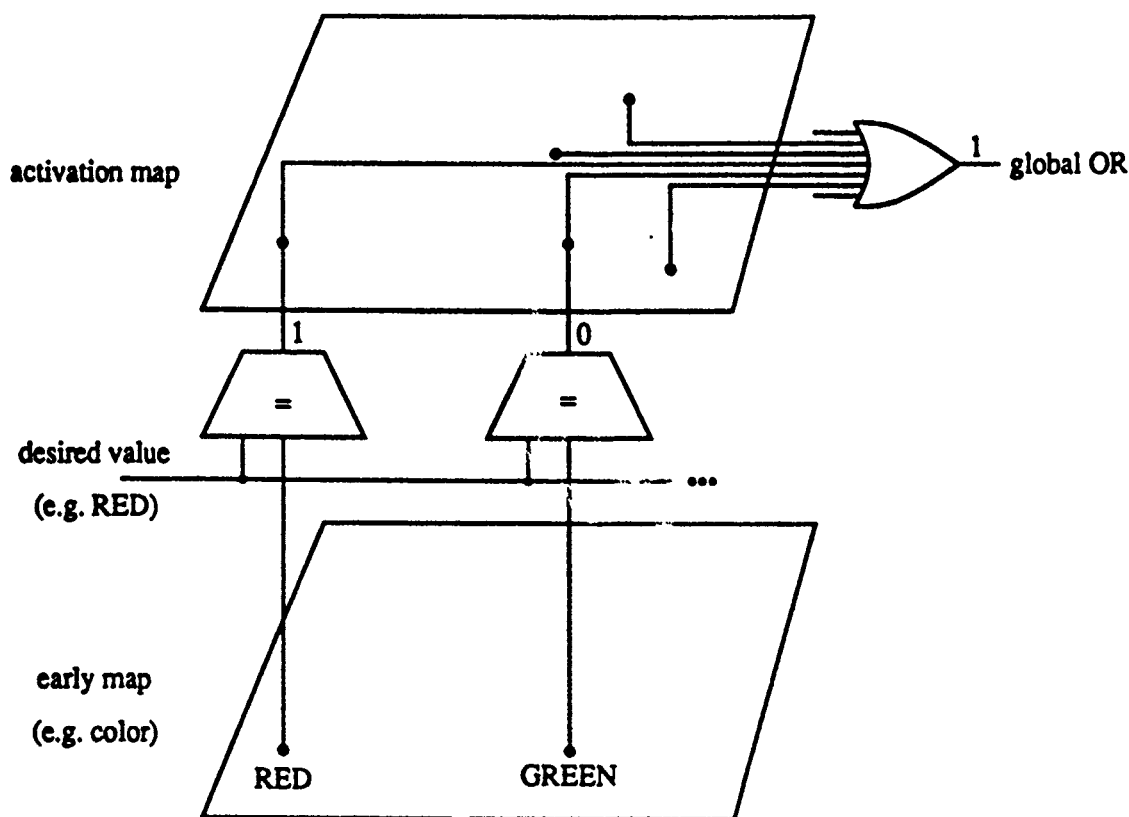


Figure 5: Structure of the activation maps. At each point the early value is compared with the desired value to give a boolean activation value. A global OR of activation values is computed over the entire map.

(which is a particular early modality) and a *value* on that dimension. Thus color is a dimension and red is a value. Each retinotopic map is retinotopically connected to an *activation map*. An activation map acts as a value filter; it has binary elements which are “on” at points where corresponding elements in the early map have the desired value.⁸ A network extending globally over the activation map distributes the desired value to all the activation elements. (An alternative implementation would use a separate activation map for each early value. This would correspond to *value unit encoding*, which seems to be the rule for cortical neurons [3].) Another global network computes the global OR (figure 5).⁹

of neurons in successively later areas of the visual cortex.

⁸Activation maps are not part of Treisman’s original model, but seem necessary to avoid searching blank areas. Similar mechanisms have been proposed previously [24, 62].

⁹Alternatively, as Treisman and Gormican [55] have suggested, the global OR could be computed using the addressing pyramid by adjusting the diameter of the attended area to span the entire visual field.

This much machinery is sufficient for tasks that require deciding whether or not there is a location in a scene which has a particular early property, and therefore accounts for parallel visual search. What about combinations of early properties? A straightforward solution would be to provide activation maps for all possible combinations. There's a good engineering reason not to do this: there are too many combinations. Assuming that there are a dozen early maps, there would be $2^{12} = 4096$ combination maps. (Value unit encoding of the individual activation maps would increase the exponent substantially.) Since retinotopic maps each take up a significant chunk of cortex [31], this is infeasible. An alternative to this proliferation of maps is serial application of visual attention, as proposed by Treisman and others.

Serial visual search requires enumerating candidates and testing to see if they have the desired property. This enumeration must be performed under the direction of some external system, which I will refer to as the *control system* and whose internal structure is outside of the scope of this paper. Various enumeration schemes are possible; I will propose a simple one which matches psychophysical results. To enumerate candidates, you pick one of the early dimensions involved in the compound desired property and enumerate all the locations that have the desired value on that dimension. For instance, if you are looking for a vertical blue edge, you can enumerate all the vertical locations and check if they are blue or enumerate all the blue locations and check if they are vertical. Enumeration involves repeated application of two primitives, *content addressing* and *return inhibition*, which affect the way nodes in the addressing pyramid select among their subnodes. In the remainder of this section I will describe content addressing, candidate testing, and return inhibition, and show how they can be combined into an algorithm for visual search.

In content addressing, the control system specifies an early dimension and value, and the addressing pyramid routes the early properties of an arbitrary location with that value on that dimension to the root. This is accomplished via the activation maps: the control system supplies the desired value to the relevant activation map and specifies that activation map as the relevant one for content addressing. Pyramid leaf nodes *disqualify* themselves from the selection process if the corresponding activation map value is zero. Disqualification propagates upwards; an internal node is disqualified if all its subnodes are disqualified. This system guarantees that the early values routed to the root node correspond to a location in the image whose activation value is one, and thus which has the desired early value on the specified early dimension. If there is no activated element in the specified activation map, the root node itself is disqualified; this corresponds to search failure.

For purposes of visual search, the root of the addressing pyramid is connected to circuits which determine whether the early properties presented there are the desired ones. In the worst case, this would require a circuit for each of the few thousand possible combinations of features. These circuits, operating on compact encodings rather than retinotopic representations, would easily fit into a small chunk of the brain. Having just one copy of these circuits, rather than a copy at each retinotopic location, is a tremendous hardware savings. Further hardware savings can be realized by computing only those combinations of early properties that are actually of interest.

Suppose the currently attended-to location does not have the desired properties; we must

reject it and find another. *Return inhibition*, when applied, prevents the currently addressed location from being considered a candidate in future content addressing. This allows candidates to be enumerated uniquely. Return inhibition requires that each leaf node in the pyramid keep a state bit which says whether or not it has been inhibited; inhibited nodes disqualify themselves. Klein [23] and Posner *et al.* [41] present psychophysical evidence for the reality of return inhibition.

In summary, an algorithm for serial self-terminating search in the proposed architecture goes as follows.¹⁰

1. Pick one of the conjoined early properties. Set the activation map for this property's dimension to filter for this value.
2. Use content addressing to find an activated location in the image. If there is none, return, signalling failure. Otherwise, the addressing pyramid will map the early properties of the found location to the root.
3. Check whether the addressed location has the desired combination of early features. If so, return, signalling success.
4. Otherwise, inhibit return to the currently addressed location. This means that future content addressings will find different locations. Go to step 2.

Sonja makes extensive use of this algorithm in playing Amazon.

Treisman and Gelade [54] report that human subjects require about 60ms per iteration of the address, test, inhibit cycle. This corresponds to examining seventeen locations per second. SIVS's cycle time varies because it was implemented on a serial machine, but on average it examines as many locations per second.

3.3 Extensions

This section presents two extensions to the basic visual search paradigm which proved very useful in Sonja but which are only weakly supported by psychophysical evidence. The first extension allows control of the order in which locations are enumerated; the second allows attention to be directed to locations based on their positions in the image, rather than on their early properties.

Controlling enumeration order

In many cases it is useful to control the order in which visual search enumerates locations. For example, domain knowledge often can tell you roughly where the sought location is likely to be.

Koch and Ullman [24], based on the psychophysical studies of Engel [11, 12], proposed a *proximity preference* mechanism for their model of the attentional pyramid. Proximity preference makes the location selected by the next content addressing as close as possible to the

¹⁰Tsotsos presents a similar algorithm [58]. His is more complicated because it involves shape matching.

currently selected location; it can be implemented with circuitry that enhances the activity of units close to the selected unit in the winner-take-all network.

SIVS provides a related form of proximity preference. It allows the control system to choose an arbitrary point of interest and causes content addressing to proceed in increasing order of distance from this point. This mechanism could be implemented using a damped spreading activation starting from the chosen point and enhancing winner-take-all units in proportion to the proximity to that point. SIVS's implementation actually uses explicit distance-comparison circuits. The chosen point is specified using a visual marker mechanism, explained in section 5. SIVS also includes a mechanism that constrains visual search to a specified region of the image or to locations lying along a specified line. In the latter case, locations may be enumerated in order along the line.

These enumeration order extensions to visual search are based solely on efficiency considerations. The only relevant psychophysical evidence I know of is due to Kröse and Julesz [26], who show that proximity preference does not *always* apply; this does not rule out its selective application under external control. It would be easy to do experiments to discover whether the human visual system has similar mechanisms. If not, people must do exhaustive searches in situations in which SIVS does not; this would result in somewhat different attentional performance.

Pointer addressing

In addition to content addressing, SIVS supports *pointer addressing*. In pointer mode, the control system can direct the pyramid to address an arbitrary (x, y) retinotopic location. This requires passing addresses *downward* through the pyramid and inhibiting nodes not addressed. The pyramid can also pass addresses *upward*, providing the control system with the image coordinates of a location addressed by content.

There is little psychophysical evidence bearing directly on the question of whether the human visual attention apparatus supports pointer addressing; the question has not been asked explicitly before. The most relevant studies ask whether or not people can direct attention to points defined indirectly, for example as "two inches to the left of the big X". Some experiments have been done along these lines, but the results are inconclusive. Kröse and Julesz [26] present evidence that argues against such addressing; Posner *et al.* [40] and Nakayama and Mackeben [36, experiment 2] present evidence that argues for it. Pylyshyn [43] argues against it on *a priori* grounds.

Whether or not an attentional mechanism supports pointers affects possible implementation strategies for other sorts of visual machinery. For example, consider the problem of determining whether one location in an image is to the left of another. An architecture that supports pointers can subtract x coordinates to answer this question; an architecture without pointers must do something more complicated.

3.4 Related computational work

So far as I know, SIVS is the first implemented system to model the phenomena described in the psychophysical visual search literature I have discussed.

Several other researchers have presented implementations of visual attention. These implementations vary in their motivations, in the faithfulness with which they model psychophysical results, and in various engineering parameters. Among the last are the type of routing network used (retinotopic, hierarchical, or all-points), selective enhancement of signals from attended locations versus selective inhibition of signals from non-attended locations, and whether or not regions of variable diameter can be addressed.

Feldman and Ballard [17] provided the first suggestion I have found for a computational implementation of covert attention. They intended both to model Treisman and Gelade's [54] psychophysical studies and to solve the connectionist crosstalk problem. They suggested using a winner-take-all network; their discussion is abstract and apparently the suggestion was not implemented. Koch and Ullman [24] similarly did not implement their proposed pyramid.

The earliest implementation I have found is due to Fukushima [19]. He used a hierarchical winner-take-all network of connectionist units. His implementation seems to have been motivated principally by engineering concerns; it does not try to model psychophysical results accurately. For example, the attended subset of the image does not need to be contiguous.

Strong and Whitehead [51] present an implemented model of *overt* visual attention inspired by Feldman and Ballard's work and similarly intended to solve the crosstalk problem.

Mozer's [33, 34] implementation of covert attention models psychophysical results better than Fukushima's; it can attend only to a single contiguous region. His winner-take-all network is not implemented hierarchically (as a pyramid) but rather retinotopically. Koch and Ullman argue that a hierarchical organization results in faster convergence of the winner-take-all computation than would a locally connected network such as Mozer's. Mozer implemented addressing of continuously variable diameter regions; SIVS doesn't. Mozer's network operates by selective enhancement of signals from the attended region, rather than by selective inhibition of signals elsewhere (as does Fukushima's implementation and SIVS). Neurophysiological results suggest that the primate attentional system operates by selective inhibition.¹¹

Ahmad and Omohundro [1] describe an implementation with a contiguous, variable diameter spotlight and boolean inhibition. This implementation is able to gate signals from the attended location to a central node in constant time by connecting the central unit to every unit in a retinotopic array. This seems biologically implausible; SIVS uses a $\log(n)$ depth fan-in tree instead.

¹¹ Moran and Desimone [32] found that neurons in area V4 of the visual cortex whose receptive fields (RFs) include the attended location respond strongly to stimuli at this location and weakly elsewhere in the RF, but neurons whose RFs did not include the attended location responded strongly to stimuli anywhere in the RF. Tsotsos [58] argues that selective inhibition should make the winner-take-all network converge more quickly.

3.5 Open questions

Many empirical and engineering questions concerning this architecture, beyond those posed earlier in this paper, remain open.

Current psychophysical evidence does not answer many questions concerning return inhibition. For example, is it applied automatically and uniformly, or selectively under control of the control system? SIVS allows the latter. How are locations uninhibited? SIVS provides a global inhibition reset line which uninhibits all locations, providing a clean slate for a new search. Perhaps individual locations can be uninhibited, or perhaps inhibition just decays over time. Is there a limit on how many locations can be inhibited? What is the spatial resolution for inhibition? It cannot be the case that enormous numbers of locations can be inhibited with great precision, or it would be easy to count patterns of many dots in arbitrary order.

Just what are the "locations" which the attentional spotlight looks at, content addressing finds, and return inhibition applies to? A simple hypothesis is that locations are the regions of the image found by a general-purpose preattentive segmentation process that partitions the image into relatively homogeneous regions. There is much psychophysical evidence that at least a first-pass segmentation is performed bottom-up [50, 53]. Psychophysical studies on attention have usually controlled out segmentation by using as stimuli displays of small geometrical figures neatly separated by a featureless white background, so little is known about interactions between attention and segmentation. More study of this interaction would be valuable; recent studies by Driver and Baylis [10] and by Farah *et al* [15] support the hypothesis that attended locations are preattentively segmented regions.

4 Visual routines

Visual routines are time-extended patterns of visual processing. Many visual properties, particularly topological properties such as connectedness and containment, are difficult or impossible to compute using a single type of processing. Different sorts of processing must be applied in sequence. Ullman's visual routines paper [60] proposes that patterns of visual processing be thought of as programs, or routines, whose primitive operations are parameterized types of visual processing. The strength of this idea is that a small set of visual primitives can be recombined into an infinite number of types of visual processing. The demands of visual analysis are very diverse; yet given the right set of operations, it may be possible to assemble visual routines capable of performing whatever sort of visual work is necessary for a new task. A vision system can thus be thought of as a sort of programming language.

As an example, Ullman describes a visual routine for computing whether or not a particular point is contained within a closed curve in the image. This routine involves applying two primitive operations. First, a "wave of activation" is propagated in the image, starting from the point of interest, and expanding in parallel in all directions, but stopping when a boundary is reached. (See figure 6.) This computation can be performed efficiently on a parallel two-dimensional grid machine. Second, a "point at infinity"—any point that is for some reason guaranteed to lie outside any curve—is tested to see whether it has been activated. If it has,

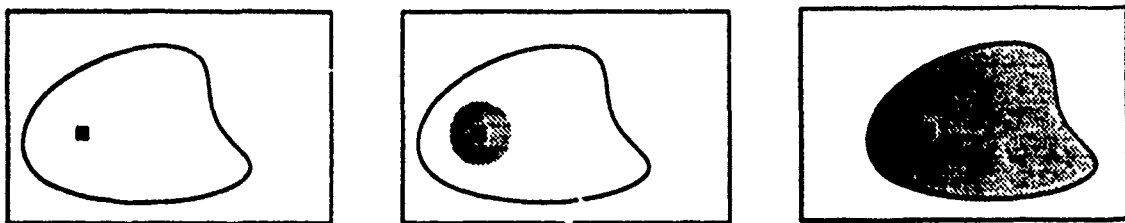


Figure 6: A visual routine for computing containment. Starting from a point marked with a solid square (first picture), a wave of "activation" is spread (second picture). The wave stops when it hits a boundary. Then, a "point at infinity" is tested to see whether it is activated (third picture). In this case, the point at infinity is marked with another square; it is not activated, and so the first square must have been inside a boundary.

we know that the activation has "leaked out" of any surrounding curve, and that the original point is not in fact contained in a closed loop. If it hasn't, we know that there is a containing curve.

In SIVS, primitive visual operations are computed by *visual operators*, which are thought of as specific pieces of neural hardware. Primitive operations and operators correspond one-to-one: although it is logically possible that individual pieces of visual hardware could compute several distinct operations, the proposed primitive operations are sufficiently dissimilar that it seems more likely that functions are allocated statically.

In the most general case, illustrated in figure 7, a visual operator takes as input zero or more retinotopic maps and zero or more control signals which determine the parameters of the operation and whether or not it is actually carried out. The operator encodes its results on zero or more compactly-encoded output buses. The operator may maintain state, typically shared with other operators; I will explain the form of this state in section 5.

Collectively, the set of visual operators constitute a *visual routines processor* (VRP). The operation of this VRP is directed by an external control system, probably the same one involved in visual search, whose nature I will again leave unspecified. The interface between the VRP and the control system consists of a fixed set of compact buses, with queries and commands as inputs to the VRP and results as outputs. This organization is similar to that of a horizontally microcoded computer (figure 8). The VRP plays the role of the datapaths; the control system is analogous to the computer's control logic. As in some recent horizontally microcoded architectures, there are many datapaths all of which operate in parallel on every cycle (though some of them may not do anything useful, if there is no computation of their sort to be performed on that cycle). Thus visual routines are strictly *patterns* rather than simply sequences of visual operations: several operations may occur simultaneously. On each tick the control system computes the parameters controlling the visual operators; this is analogous to the vector of control bits provided by a horizontal microinstruction.

Depending on the primitive operations selected, on task requirements, and on the mech

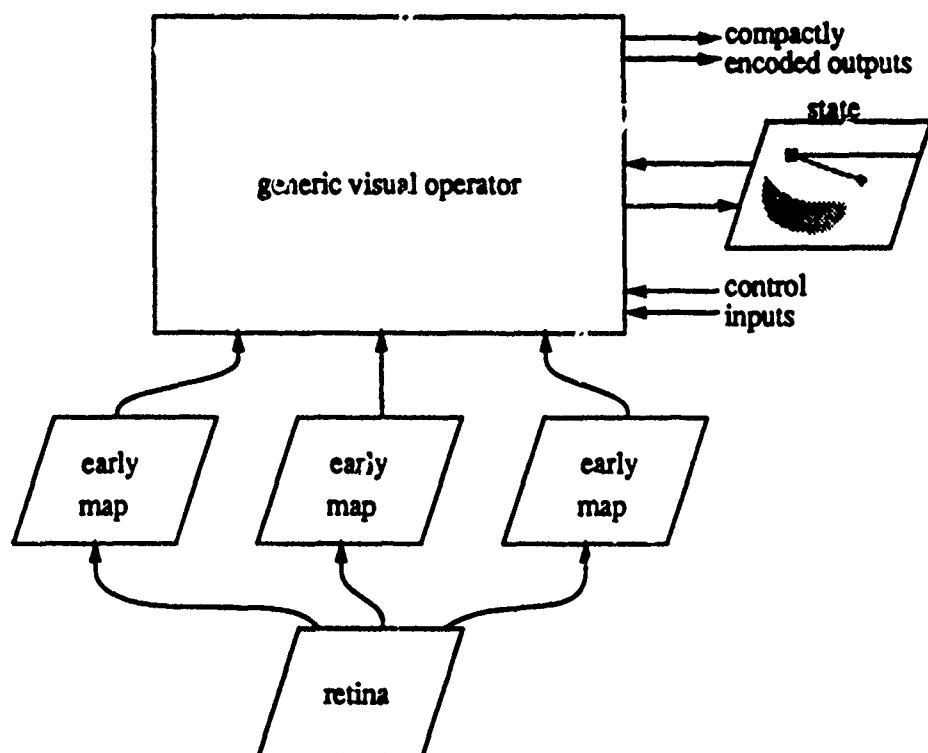


Figure 7: A generic visual operator. The visual operator, on the basis of control inputs, produces compact outputs from the retinotopic maps. It may maintain some state, which can be shared with other visual operators.

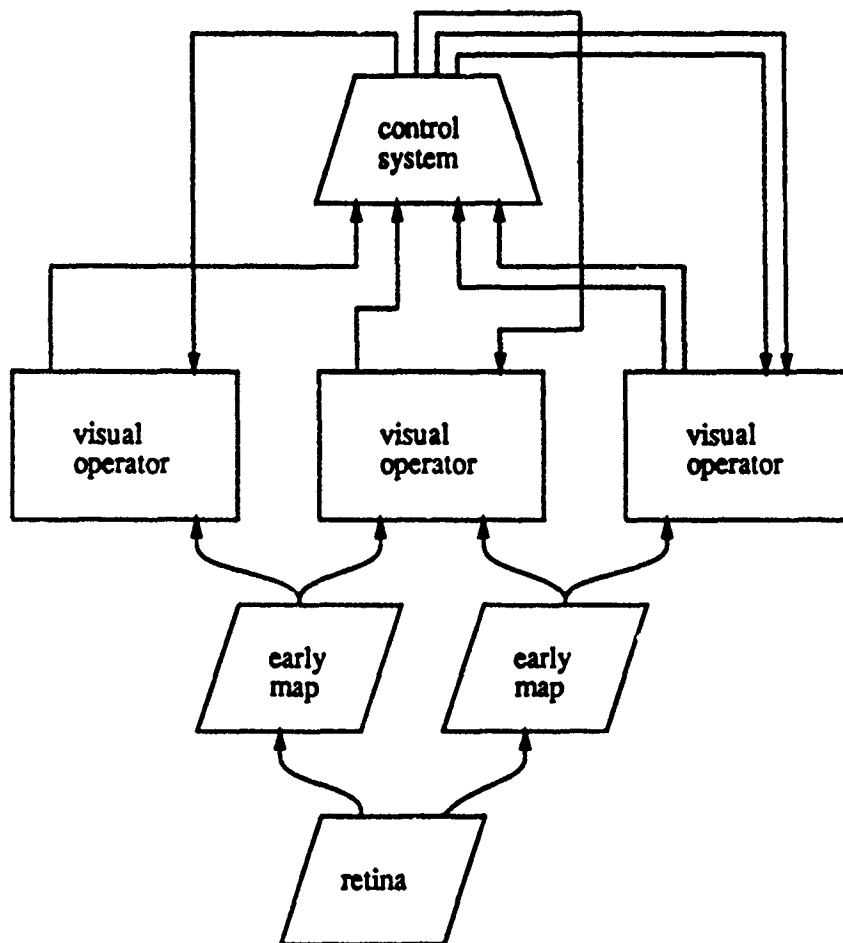


Figure 8: Overall modularity. A control system takes inputs from the visual operators and produces outputs for them.

anisms that determine when operations are performed, visual routines might be used in very different ways. For Sonja, visual routines are a means for *task-specific, top-down* control of visual processing *in support of action*. "Task-specific" means that visual routines are designed to discover properties of domain situations that are meaningful in terms of the task the agent is engaged in. "Top-down" means that visual routines are invoked on the basis of factors other than just the currently visible image, such as the memories and purposes of the agent. In such a model, you might have task-specific routines for checking your speedometer, for glancing at your keyboard to align your fingers in home position, for checking a pancake to see if it is ready to flip, and for finding safe footing when walking in the mountains. These tasks are subtasks of other tasks, which are not purely visual, but which are guided by visual feedback. In section 7 we will see a complex example of a task-specific visual routine that guides activity in Sonja.

These commitments to task-specificity, top-down control, and visually guided activity are not necessary correlates of the visual routines architecture. Visual routines might instead be used as part of a bottom-up object recognition system, for example, or they might be applied uniformly and might deliver purpose-independent representations as outputs. SIVS's architecture might carry over directly to such applications although that is not the way I have used it.

I will postpone discussion of the specific operators in SIVS until section 6.2, after explaining the data structures the operators use (described in section 5) and the criteria used in choosing the operators (described in section 6.1). However, their purpose, in general terms, is spatial analysis: establishing both geometrical and topological relationships between portions of the image.

5 Intermediate objects

We saw in section 4 that visual operators can access state variables. This state is required to keep track of intermediate results during visual routines. We have seen one example already: the activated region that is computed in the first step of the containment routine (figure 6). Lacking empirical constraint on the nature of these intermediate representations, I have adopted Ullman's proposals, which were based on computational intuition, and extended them based on my own computational intuition.¹² These proposals might be tested psychophysically.

SIVS's visual operators manipulate four types of *intermediate objects*: *markers*, *lines*, *rays*, and *activation planes*. These representations are shared across operators, rather than being private to particular ones. Visual markers designate locations in the image. Lines (actually directed line segments) run between two points; rays extend from a point to infinity. Activation planes represent regions in the image. In figure 17, for instance, we see some markers, lines, rays, and activated regions displayed graphically on top of a running Amazon game. The reverse-video polygons represent markers, the drawn lines represent line and ray intermediate objects, and the shaded regions represent activation planes. (These graphical representations

¹²The only previous implementation of visual routines, due to Romanycia [45], used only retinotopic representations.

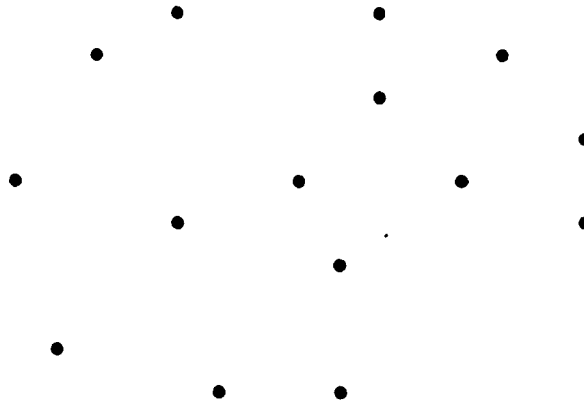


Figure 9: To determine whether there are four colinear points, you have to keep track of points to apply the colinearity test to.

should not be confused with the intermediate objects themselves.) All four sorts of intermediate objects are image-centered, representing only two-dimensional information. Three-dimensional processing is beyond the scope of this paper; see [7] for ways to incorporate it.

The interface between the control system and the VRP is in part in terms of the intermediate objects. The control system can name intermediate objects; that is, it can pass compact encodings which say which marker, line, ray, or activation plane to use in an operation. Operators can, for instance, determine whether the distance between one pair of markers is greater or less than the distance between another pair; draw a line between two markers; or determine whether a marker is within an activated region.

The remainder of this section describes in turn markers, lines and rays, and activation planes.

Markers

Many visual operations require storing locations. For example, if you want to know if there are four colinear points in an image, you have to keep track of points to apply the colinearity test to (figure 9). Visual markers are one mechanism for keeping track of locations. The simplest implementation of location stores would be registers holding image coordinates. Since stored locations are typically found using visual search, this implementation requires that the addressing pyramid be able to pass addresses in at least the upward direction.

Visual markers are not intended as a complete theory of the ability to store locations. Markers can only represent visible objects, whereas people can remember the locations of objects that are no longer visible. Psychological evidence has led several researchers [16, 42, 43] to propose that there are several distinct spatial memory mechanisms; markers might be one.

Lines and rays

Lines and rays in SIVS may not correspond to any "things" in the human visual system. They were an easy interface for various useful visual operators which may well use some other interface. Their principal use is to specify spatial limits for a search: SIVS has operators that find things that lie along a straight line or a ray. Whether such limits can actually be put on visual search is unknown but could readily be determined experimentally. One way to implement them would be to "draw" the line on a retinotopic map that is ANDed into a search activation map. Then only locations lying on the line would be candidates for content addressing. Another implementation would scan attention serially along a line. These implementations could be distinguished psychophysically by reaction time data.

Activation planes

Activation planes are used to keep track of interesting regions of the image, as in Ullman's routine for computing containment. They can be naturally implemented as retinotopic bit arrays, one bit array per plane; bits are turned on at points that are within the region of interest.

Psychophysical evidence could help support or disconfirm the existence of activation plane hardware. I know of only one relevant study, due to Farah [14], who found that imaging a complex bounded form increased the detectability of events within the bounded region. This effect was found to be similar to attending to a colored form of the same shape. An activation plane would be a natural mechanism underlying this effect. Kosslyn has suggested an experiment (described by Ullman [60] but apparently never performed) that would give a more direct test. In it facilitation of later inside/outside judgements by a first judgement would suggest the existence of a representation of the extent of the bounded region.

How many activation planes are there? I know of no psychophysical studies of this question. The following informal observation may serve as the basis for an experiment. When staring at floors tiled with a regular pattern of identical tiles, I find that I can cause specific subsets to jump out: a hollow or filled square or hexagon (depending on the tiling pattern) or, more interestingly, disconnected sets like alternate tiles (resulting in a checkerboard appearance). This phenomenon is quite striking in that I can make quite elaborate patterns appear globally over a space of hundreds of tiles. The jumping-out tiles almost literally appear to be darker or "colored." If this is the phenomenological correlate of activation, it suggests that there is only one activation plane available for this purpose, because despite much effort, I am completely unable to form even simple patterns that divide the surface into three sets rather than two. Simple psychophysical experiments might decide this question. In any case, Sonja uses three activation planes, but could probably get by with timesharing a single one.

The next section explains how intermediate objects are manipulated by visual operators.

6 Visual operators

Visual operators are to be thought of as bits of hardware each dedicated to performing a specific visual operation. Because visual operators support intermediate vision, their purpose is to give compact answers to compact questions about non-compact representations such as image regions and activation planes. (Recall figure 7.) Because the range of visual processing tasks is so broad, there are many visual operators with distinct functions. Individually they may not do much, but they may be combined by serial application into powerful visual routines. These routines involve partial results which are kept track of with intermediate objects.

Section 6.1 explains how we might choose and evaluate a set of visual operators; section 6.2 describes the specific set implemented in SIVS. Section 7 provides examples of the combination of these operators into visual routines.

6.1 Criteria on visual operators

There are two sorts of criteria that bear on choosing perceptual operators: local criteria on individual operators and global criteria on the set of them.

The local criteria I used in designing SIVS were that an operator be implementable in biologically plausible hardware, general purpose, neurophysiologically and psychophysically plausible, and clean from an engineering standpoint.

- Each operator ought to be implementable in biologically plausible hardware. The local connectivity and slow processing speed of neural hardware imply that visual computations (which typically operate in less than a second) must involve no more than about a hundred sequential steps [17] and make certain mechanisms such as pointers expensive—perhaps prohibitively so. Ideally SIVS would implement each operator as a network of neuron-like units. However, even the most powerful parallel computers available today would not have been up to the job of simulating the number of units required, and so I implemented most operators with conventional serial algorithms. However, I will sketch massively parallel, pointer-free implementations for each visual operator, thereby arguing that the constraints of biological plausibility do not rule them out *a priori*.
- The operators should be general purpose in two senses: they should not depend on the specific domain in which the system is tested, and they should be useful for very different sorts of tasks. It is impossible to be sure without doing cross-domain and cross-task studies, but my intuition is that all SIVS's operators satisfy this criterion.
- Each operator's membership in the set ought to be supported by neurophysiological and psychophysical evidence. This is not true in SIVS. Most of the relevant experiments have not been done. Many of the operators in SIVS suggest tests for comparable human performance.
- Finally, I used straightforward engineering considerations to choose many of the operators. I used programmer's intuition to judge whether the postulated operators involved seemed

clean. SIVS includes only one inelegant operator (explained in section 7), in a case in which "doing the right thing" seemed like it would be a lot of work and not particularly edifying. I am reasonably sure that this operator does nothing that could not be done cleanly.

My global criteria on the set of operators were that the set span the space of visual analysis tasks; that it make programming visual routines easy; and that it make learning visual routines easy. I will postpone evaluating SIVS according to these criteria until section 8.2.

- We want a "spanning" set: that is, a set of operators that together are sufficient for any task. Here "any task" may mean "any psychologically plausible task" or, for engineers, "any task in the class of domains of interest." Thus the set of visual operators, when combined into routines, are to form a finite means for the realization of an infinite collection of possible visual processes.
- A set of operators should not only make it possible to implement any visual task, it ought to make it *easy*. From an engineering standpoint, the VRP should present a nice programming system.
- A set of operators should also make it easy to *learn* new visual routines.

6.2 SIVS's visual operators

This section describes the specific visual operators SIVS uses. This set substantially extends the set proposed by Ullman [60].

The subsections of this section correspond to a somewhat arbitrary categorization of operators into six groups. The first group is concerned with visual attention and search; the corresponding subsection, 6.2.1, fills in some details left vague in sections 2 and 3. Subsection 6.2.2 describes an operator for tracking moving objects. 6.2.3 describes operators concerned with geometry: distances, directions, and angles. Subsection 6.2.4 describes operators for direct manipulation of intermediate objects and 6.2.5 describes operators involving activation planes; together these can be used to determine topological properties such as containment and connectedness. Subsection 6.2.6, finally, describes a housekeeping operation called "blanking."

By convention, the names of operators that change the state of intermediate objects end in an exclamation mark (*content-address!*), and those that implement boolean tests end in a question mark (*distance-within?*).

6.2.1 Visual attention and search

I have explained the implementation of visual search in section 3; this section explains the details of the interface between it and the control system. This interface is in terms of visual operators which encapsulate the addressing pyramid, thereby integrating it into the architecture of figure 8.

Attention supplies early properties to the control system. I gave the VRP as plausible an interface with the control system as I could, but made no attempt to make the connection

between early and intermediate vision realistic. SIVS does no pixel-wise early processing. Instead, the visual operators have direct access to the data structures representing video game objects that Amazon maintains for its own purposes. I will talk about the implications of bypassing early vision in section 8.1.

Thus, to implement content addressing, I needed a simulated implementation of early processing. This implementation is in terms of seven early dimensions. I chose these dimensions and the values assigned to video game icons along these dimensions somewhat arbitrarily; my main concern was to ensure that some objects would pop out and that others would require slow serial searches. Some of the dimensions are found in human early processing: grey level, speed and direction of motion, line orientation, and perhaps overall size. Two others are arbitrary and probably not biologically accurate: I called them "fiddliness," corresponding roughly to the amount of detail in the icon, and "boxiness," corresponding to whether or not the icon is roughly rectangular. Early properties are *not* computed at run time, but are fixed properties of icons. The implementation does not model the internal structure of the icons; they are treated as homogeneous blobs. SIVS takes icons to be the "locations" to which return inhibition applies; see section 3.5.

The operator `content-address!` implements content addressing. It takes as inputs an enable signal and an early (dimension, value) pair to find. Because one typically wants to keep track of the addressed location, the operator takes a marker as an additional input; the marker is moved to the center of the addressed location. Optionally `content-address!` can take as input another marker representing the locus of proximity preference (section 3.3).

Three operators put spatial limits on content addressing. `content-address-activated!` requires that the found location be within an activated region. `scan-along-line!` and `scan-along-ray!` address the first location with a given early property found along a line or ray.

The operators `inhibit-return!` and `uninhibit-return!` perform the operations they are named for and take only an enable signal as input.

6.2.2 Tracking

An ability to *track* several moving objects simultaneously is very useful in playing video games. My informal observations of human players suggest to me that people can track up to four or five moving things simultaneously; Pylyshyn and Storm [43, 44] present psychophysical evidence for about the same numerical limit on tracking.¹³

SIVS tracks moving objects with visual markers. The operator `track!` causes a marker, passed as an input, to track the motion of the thing it marks. As many copies of `track!` are required as things can be tracked simultaneously. SIVS supplies five copies, of which Sonja uses only three. SIVS also provides `disappeared?` operators which tell whether a tracked thing has been lost.

¹³Pylyshyn argues that this ability contradicts the finding that people can attend to only one location at a time. No such contradiction is necessary, however, if attention is required to support access to *all* early properties. Full access is not necessary for tracking, which needs access only to motion computations. The tracking hardware probably has a separate, dedicated addressing scheme for a motion map.

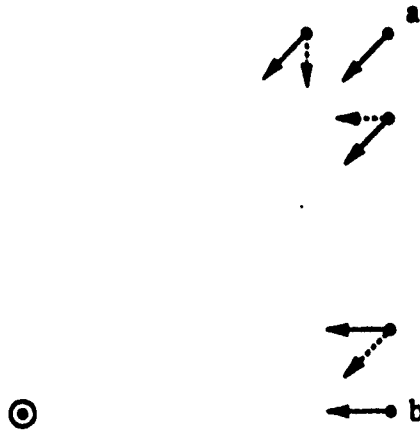


Figure 10: Major and minor components of directions to the circled point. The major component is drawn as a solid arrow and the minor component as a dashed one. In two cases (*a* and *b*) the two components coincide.

In the human visual system, tracking presumably works by segmenting the optical flow field. How this works in detail is unclear (see Thompson and Pong [52] for an explanation of the issues and some computational approaches) but there is no doubt that the human visual system supports tracking as a primitive [46].

6.2.3 Distances, directions, and angles

SIVS provides a series of operators which compute geometrical relationships between intermediate objects. These operators manipulate distances, directions, and angles. All of these operators are implemented as numerical computations in terms of the pixel coordinates of intermediate objects. Directions are coded as ordered pairs of the eight "king's move" directions, a *major* and *minor* component representing the nearest and next nearest of the eight directions to the actual direction. (See figure 10.) This encoding provides sufficient resolution for the purposes to which the system is put.

The operator *distance-within?* is a predicate on two markers and a distance; it tests whether the distance between the two markers is greater or less than that given. *greater-distance?* takes two pairs of markers and tells whether the distance between the first pair is more or less than the distance between the second. *markers-coincident?* tells whether the distance between two markers is zero.

marker-to-marker-direction tells the direction from one marker to another. *aligned?* tells whether or not two markers are aligned in one of the eight directions.

Three operators, *angle-ccw?*, *marker-line-angle-ccw?*, *marker-ray-angle-ccw?* determine

whether or not an angle is counterclockwise; they respectively take three markers, a marker and a line, and a marker and a ray as inputs.

Two operators determine whether locations (interpreted as Amazon icons) are adjacent to each other. `touching?` takes as inputs a marker and a direction; it tells whether or not the Amazon icon under the marker is touching something else on the specified side, or whether there is free space in that direction. `corner-free?` similarly determines whether there is free space adjacent to a given corner of an icon. These operators probably ought to be decomposed into routines over more primitive operators that would shift attention to the indicated edge of the icon and check for free space.

6.2.4 Marker, line, and ray manipulation

Two operators provide direct manipulation of marker positions. `warp-marker!` moves one marker to be coincident with another. `walk-marker!` moves a marker a specified distance in a specified direction. These are implemented by side-effects to the coordinate information that markers maintain. Similarly, `draw-line!` takes two markers and a line and causes the line to extend from one marker to the other, and `draw-ray!` extends a given ray from a given marker in a given direction.

6.2.5 Activation

The primary use of activation planes is to fill (activate) a bounded region. Ullman [60] suggests that what counts as a boundary may be task-specific (and thus is presumably supplied as a parameter by the control system). Lacking evidence about the nature of this information in the human visual system, I had the operator take a disjunction of icon types that are allowed in the activated region. Ullman suggests further that short gaps in surrounding edges may optionally count as boundary segments for the activated region, and SIVS provides an optional gap filling facility.¹⁴

Region filling is implemented by the operator `activate-connected-region!`, which takes a marker to start spreading activation from, a plane to mark the region with, and the type information about what will count as boundaries. The operator returns a single boolean value, which tells whether activation succeeded or if the spreading would, rather, continue to infinity.

Activation can be efficiently implemented in a locally-connected retinotopic array of processors [60]. You turn the `activated?` bit on in the processor corresponding to the marked point from which activation begins, and then you repeatedly propagate activation: each activated processor tells all its neighbors that it is activated; if they are boundary points they do nothing; otherwise, they set themselves activated. Repeat until no new processor becomes activated. Mahoney [29] and Shafrir [47] describe still faster divide-and-conquer activation algorithms.

Three operators, `marker-activated?`, `line-activated?`, and `ray-activated?` determine whether other intermediate objects intersect activated regions. Operations like these could be

¹⁴Psychophysical study of gap filling in edge tracing, postulated as a related operator, is reported by Jolicoeur *et al.* [21]. Ullman [59] suggests that gap filling is a ubiquitous operation in early vision and proposes neural networks for the operation.

implemented by connecting corresponding elements in the retinotopic arrays representing the various sorts of intermediate objects. For instance, in section 5 I have suggested implementing lines by "drawing" them on a retinotopic array of computational elements; *line-activated?* could be implemented by having these elements communicate with corresponding elements in the activation plane to determine if they represent a point that is both activated and on the line; computing a global logical OR over the array will yield the desired result.

SIVS provides several other operators that manipulate activation planes. *mark-centroid!* puts a marker at the centroid of an activated region. A biologically plausible implementation might use a spreading activation computation [18]. *convex?* tells whether a activated region is convex, and *expand-to-convex-hull!* takes two activation planes and makes one be the convex hull of the other. Computing convex hulls is probably not psychologically realistic, but for the purpose I put it to, a realistic and equally useful operation would be to compute a Gaussian blurring of a region; such blurring operations have been demonstrated neurophysiologically in human early vision [30].

transect-activation! takes two activation planes, a line, and a direction. It sets one of the activation planes to be the subset of the other plane that is on the side of the line indicated by the direction. (See figure 17 for an example.) One way to implement transection would be in terms of the activation propagation algorithm previously described; it would require turning on the *boundary?* bits in elements corresponding to locations along the line.

6.2.6 Blanking

Blanking is a sort of housekeeping operation. A blanked intermediate object is unused and has no spatial information associated with it. For each intermediate object there is an operator which tells whether or not it is blanked and one that blanks it. There is no explicit way to unblank an object; operators that side-effect objects unblank them. In implementation, blankedness is just a bit associated with each intermediate object.

This concludes the enumeration of SIVS's visual operators. It is not hard to think of other operators of the same general character that could be added to the set. (For example, SIVS does not currently support boolean operations on activation planes.) This suggests that the set is incomplete; this issue is taken up in section 8.2. However, the next section will demonstrate that SIVS's operators are adequate to support complex visually guided activity in at least one domain.

7 Visually guided activity

This section describes the use of SIVS to guide action in Sonja. Sonja plays a competent beginner's game of Amazon. Its access to Amazon is only via SIVS and the game's primitive actions. Study of such visually guided action is important for several reasons:

- It is an interesting and ubiquitous phenomenon in its own right, and one that has been relatively little studied, at least within AI.

- The support of practical activity is a main function of vision, and one which seems to have different requirements from (for example) object recognition.
- Practical use of SIVS shows that the intermediate visual mechanisms described in this paper actually are useful. This does not necessarily demonstrate that these mechanisms will be useful or sufficient in other domains or that they are biologically accurate models, but it does suggest that they are worthy of further empirical and computational study.
- By coordinating the various intermediate visual processes integrated in SIVS, Sonja demonstrates that combinatorial power of simple mechanisms.
- The use of psychologically realistic vision puts significant constraints on practical reasoning and representation. For example, the bandwidth limitations of visual attention imply limitations on representation in reasoning under time pressure, and thereby rule out some but not all popular approaches to reasoning about action [6].

In Sonja, the "control system" that supplies parameters to visual operators is the same as that responsible for reasoning about action; it is described in [7]. Because this system is beyond the scope of this paper, I will devote this section to presenting an example of Sonja in operation. This example will give a sense of the ways in which many different sorts of visual operations, including search, can be combined effectively to guide action; it should not be hard to see how to make the relevant control decisions. What matters is that the control mechanisms base their choices on visual information derived from visual routines using the operators described in section 6.2. Sonja's cycle time is well under a second, engendering tight coupling between perception and action.

Before more concrete discussion, I must explain the relevant aspects of Sonja's domain, Amazon. I chose a video game as a domain in part because it allowed me to finesse both early and late vision in SIVS. Doing so is dangerous, of course; as section 8.1 points out, it may not be possible to connect a SIVS-like intermediate visual system with real early and late vision. Amazon, as a domain, has several positive qualities from point of view of vision research, however.

- It is a naturally occurring task domain, closely patterned on a popular arcade game. People regularly engage in the same task Sonja performs.
- The task is intensely visual. Action decisions can be made mainly by examining the current, visually accessible situation, without needing extensive reference to past situations or elaborate reasoning about hypothetical future worlds.
- The scene presented on the game screen changes frequently, reflecting a dynamic underlying domain, and affording opportunities for research on visual processing in the face of change.

Amazon is a dungeons-and-dragons-like game based on the commercial game Gauntlet. Figure 11 is a screen snapshot of the Amazon window as it appears to the player. The player (a person or Sonja) controls an icon on the screen representing a woman warrior: the amazon. The window actually provides a view into a small part of a larger underlying dungeon composed of bricks which form obstacles, walls, and rooms. The window tracks the amazon; when it gets

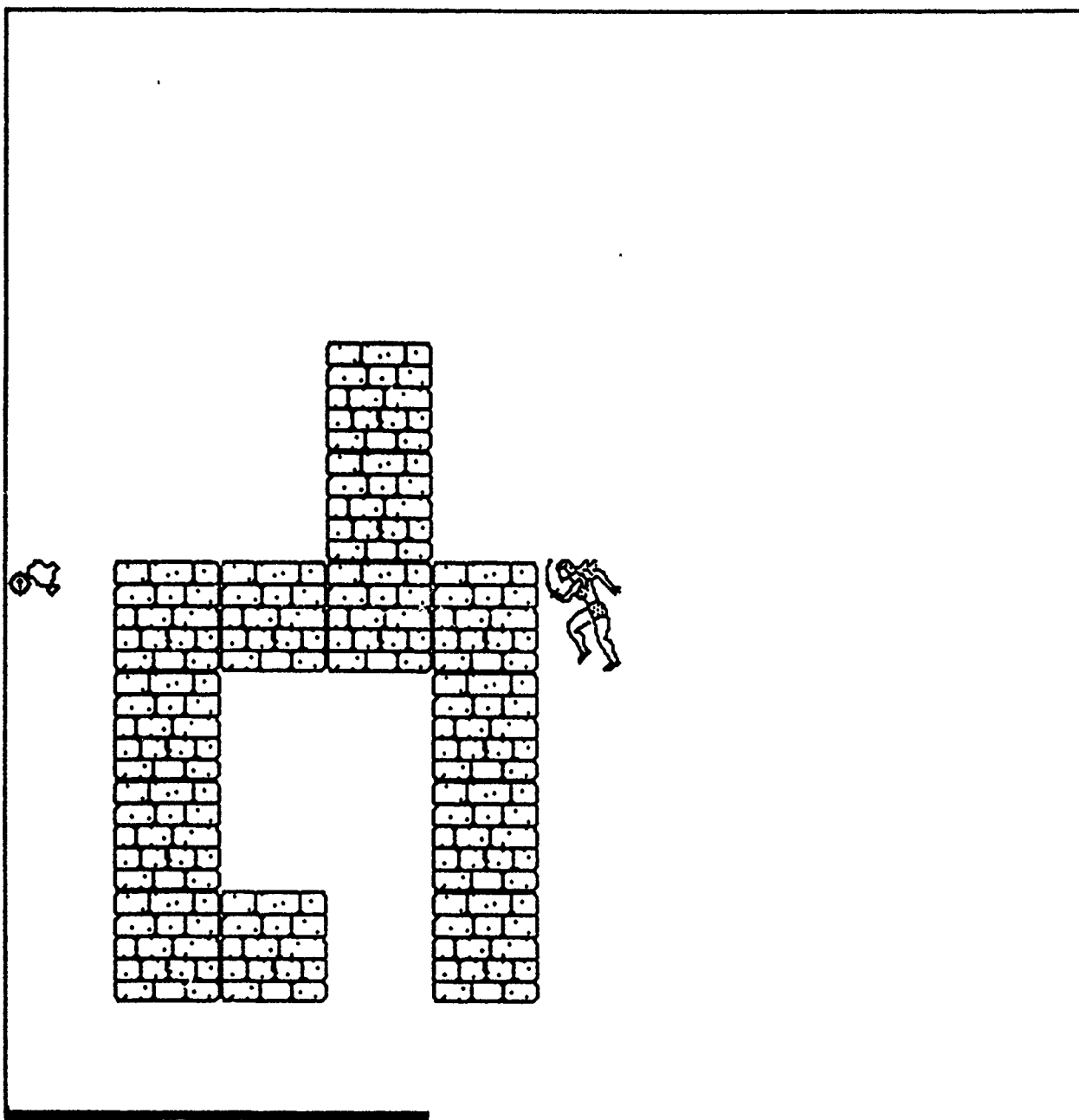


Figure 11: A simple Amazon scene. The player controls the amazon icon, which in the scenario presented in this section must navigate around the obstacle to get the amulet.

close to the edge of the screen, the window smoothly moves over the underlying scene to keep it within bounds, revealing new parts of the dungeon as it goes. The amazon can move in the eight king's-move directions only; motion is continuous (actually a pixel at a time and fast enough that there is no noticeable flicker). In the dungeon there are various sorts of enemies and tools. In figure 11, for instance, there is an amulet at the extreme left which, when picked up, confers magical powers on the amazon. There are many further complexities to the domain which are not relevant here.

The scenario presented in this section will demonstrate only a small fraction of Sonja's abilities; specifically, one of several routines for moving the amazon about in the dungeon. Sonja plays Amazon from the same perspective a human does: looking at the screen, which is to say as if looking down on the dungeon from above, not from the point of view of the amazon icon. There are well-known algorithms for navigating in mazes seen from above; depth-first search is an obvious one, and probably one can't do much better in the general case of complex and deliberately confusing mazes. I have not adopted such a solution. Amazon's mazes are simple enough that it is visually obvious how to get about in them; search would be overkill.

It seems plausible to me that a human Amazon player instead uses visual routines whose job it is to determine how to get from one point to another. Navigation thus depends mainly on continuous visual analysis of the current situation. It also seems plausible that there are several of these routines, specific for different sorts of situations. For example, different routines might analyze the scene in terms of obstacles to avoid, rooms to enter or exit, or passageways to follow. (Eye tracking studies might be used to explore this hypothesis.)

When a new Amazon game begins, SIVS is initialized, blanking all the intermediate objects. The first order of business is to find the amazon using the visual search algorithm of section 3.2. Sonja uses `content-address!` and `inhibit-return!` to enumerate things, such as the amazon, whose value for the early dimension `size` is `medium`. Sonja checks each successively addressed object to see whether it has the conjunction of early properties that are criterial for amazonhood, namely being `fiddly` and having `diagonal` elements in addition to having `size medium`. Since relatively few objects other than the amazon are of medium size, the search can be expected to terminate quickly. (In figure 11, the only object Sonja might examine besides the amazon is the amulet.) Sonja then permanently allocates a marker to tracking the amazon, using a copy of the `track!` operator. Sonja also draws a ray from the amazon in the direction it is heading, using `draw-ray!`. Figure 12 shows the outcome of this amazon-finding routine.

Once Sonja has found the amazon, it looks around the screen to find interesting objects. It does this with another visual search, specifying the early dimension `grey-level` with desired value `non-zero`, thereby enumerating all objects. There's no way to avoid enumerating everything if you want to be able to find interesting things of arbitrary types, because there is no single early property that interesting things share that is not also shared by uninteresting objects such as bits of wall. Accordingly, this search can spend many ticks enumerating bits of wall before finding anything interesting. In the example at hand, Sonja eventually finds the amulet, identifies it as such by examining its early properties, and drops a marker on it (figure 13).

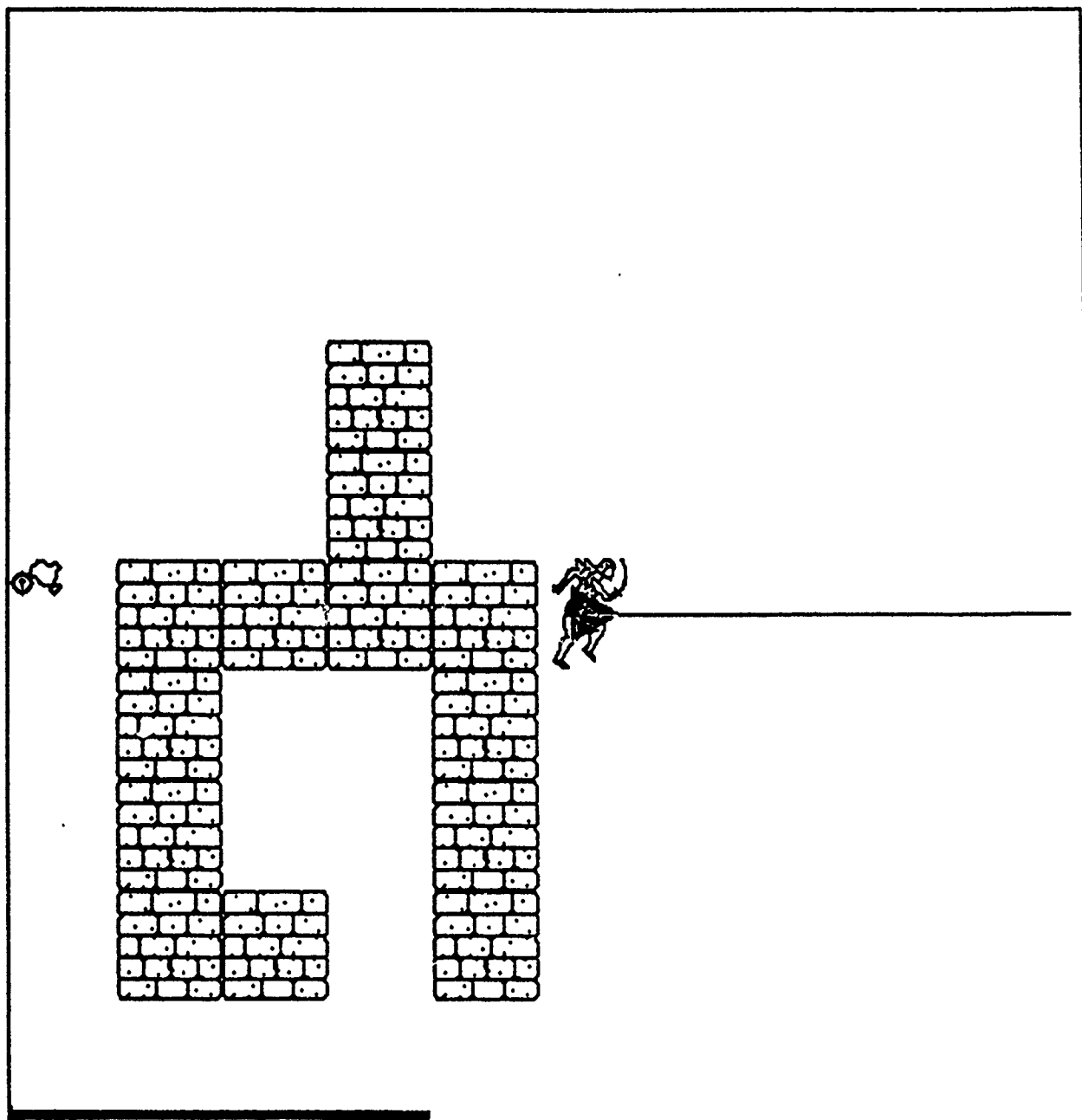


Figure 12: Finding the amazon. Sonja tracks the amazon with the right-pointing marker and extends a ray in the direction the amazon is heading. Recall that the inverse-video graphics are only representations of the intermediate objects, which are not implemented graphically.

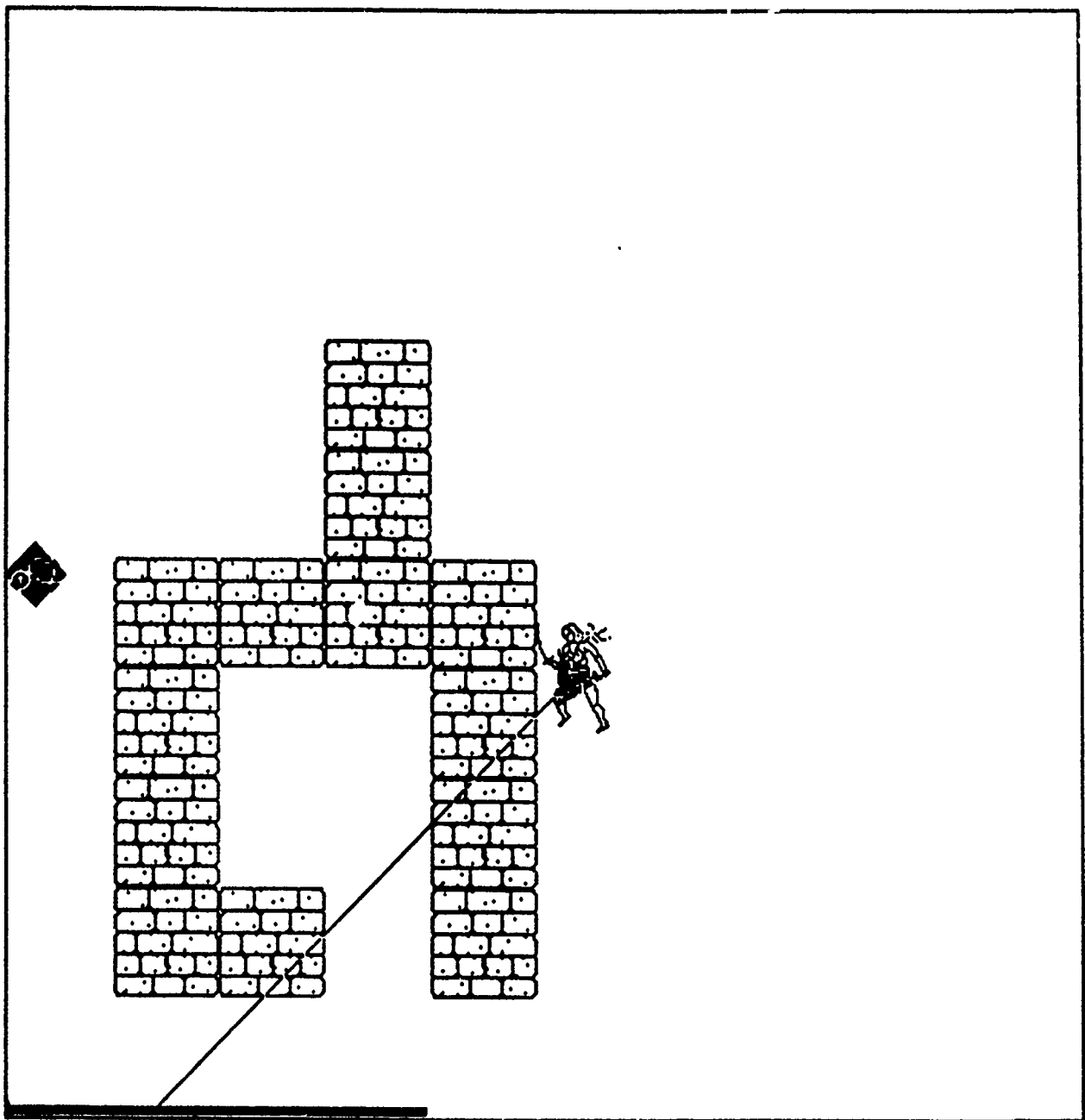


Figure 13: Sonja finds the amulet and marks it with the diamond-shaped marker.

To get the amazon to a goal (such as the amulet), Sonja first draws a line between their respective markers using `draw-line!`. Then Sonja uses `scan-along-line!` to determine whether this line intersects anything that would constitute an obstacle.

If there were no obstacle, the amazon could head directly for the goal. (Remember that the amazon moves continuously.) Sonja finds the direction to the goal using `marker-to-marker-direction` and by default heads the amazon in its major component. (The major and minor components of directions were explained in section 6.2.3.)

There is an obstacle to getting to the goal, and `scan-along-line!` drops a marker on it (figure 14). Typically, the obstacle is a largish object; simply putting a marker on it somewhere is not enough to know how to pass it. In order to discover its extent, Sonja uses `activate-connected-region!`, passing it the obstacle marker, a type specification for obstacles, and an activation plane. This activates the whole obstacle (figure 15).

Having found an obstacle to getting to the goal, Sonja has to figure out whether or not the obstacle is a room that must be entered or exited or whether it can simply be passed (as is the case).

Sonja's visual routine for determining whether or not an icon is in a room is similar to the abstract containment routine proposed by Ullman and discussed in section 4. It uses `activate-connected-region!` to spread activation outward from the icon until it runs into an obstacle boundary. The operator knows to skip over short gaps in the boundary; in this case those may constitute doorways in a room. In the scene illustrated in the figures, had the amulet been in the center of the obstacle, the walls would have acted as a room to enter, with the doorway at the bottom right. The operator `activate-connected-region!` fails if activation runs off the edge of the screen, which corresponds to the "point at infinity" of section 4. If activation extends off the screen, the goal is not bounded by a room, or at any rate not one that is currently wholly visible. In figure 15, neither the amazon nor the amulet is in a room, and `activate-connected-region!` fails for both.

If the amazon is in a room and the goal is in the same room, or if neither is in a room, then the entering and exiting code is not applicable. Responsibility for getting to the destination rests in such cases with code for passing obstacles.

The passing code has one decision to make: which way to go around the obstacle. This decision is quite complicated in general. By default, the best way around is the shortest. However, if part of the obstacle is offscreen, the apparently shortest way around may not be.

To discover which is the shortest way around the obstacle, Sonja finds the centroid of its convex hull. First it uses `exp11-to-convex-hull!` to activate the convex hull of the obstacle. Then it uses `mark-centroid!` to move the obstacle marker to the centroid of the convex hull (figure 16). Sonja can then get an idea of which way around the obstacle is shorter by examining the sign of the angle between the goal, the amazon, and the center (using `angle-ccw?`). It is usually the case that the shortest way around has the same sign as this angle. In the figure, for instance, the shorter way around is counterclockwise.

Next Sonja needs to determine whether the obstacle extends offscreen in the direction it would pass it if it took the apparently shortest way around. If so, it would typically be better to go the other way, because the obstacle might extend arbitrarily far offscreen. (Recall that the

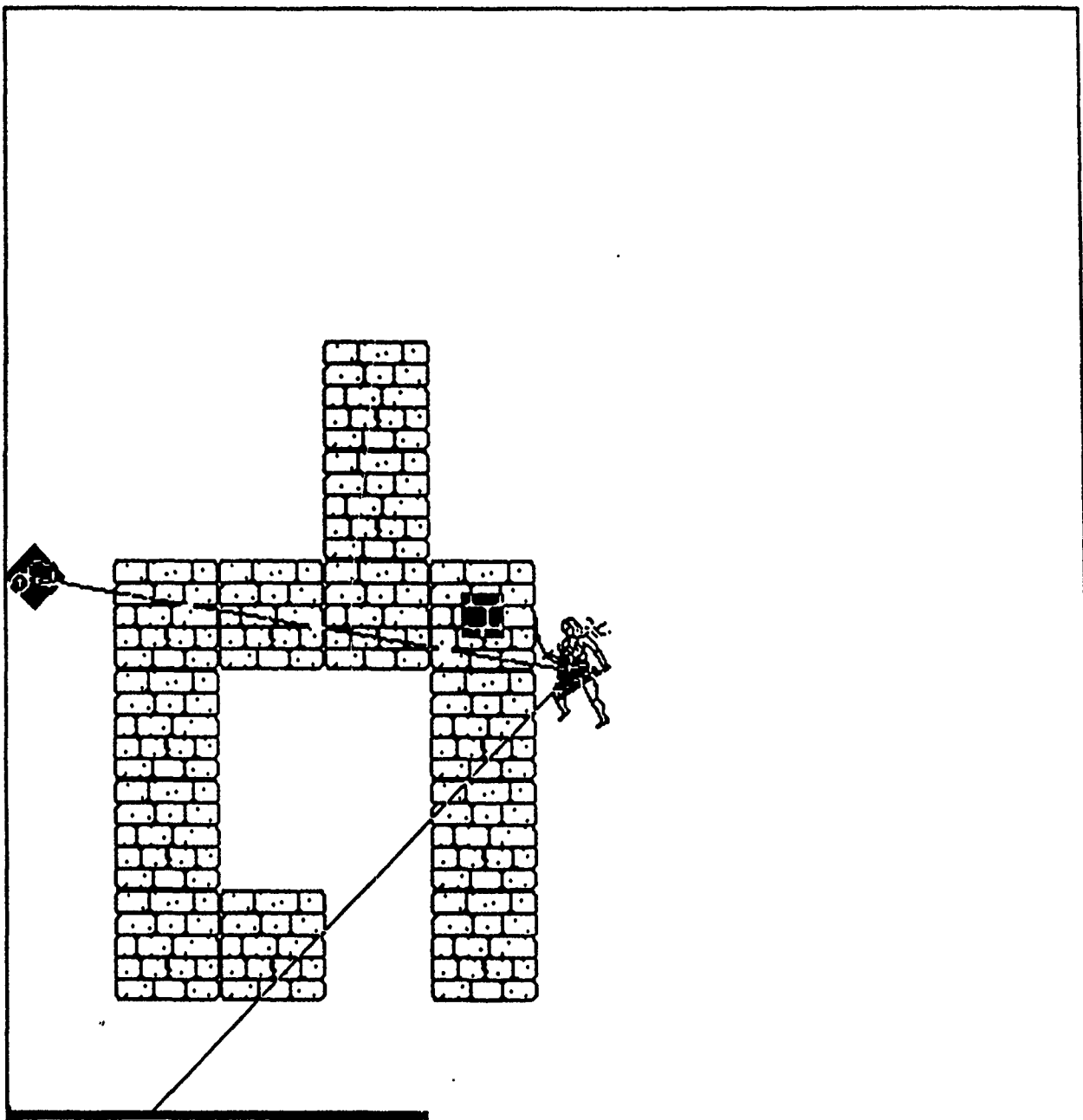


Figure 14: Marking the obstacle by finding the first thing along the line between Sonja and the destination. The obstacle marker is the square one.

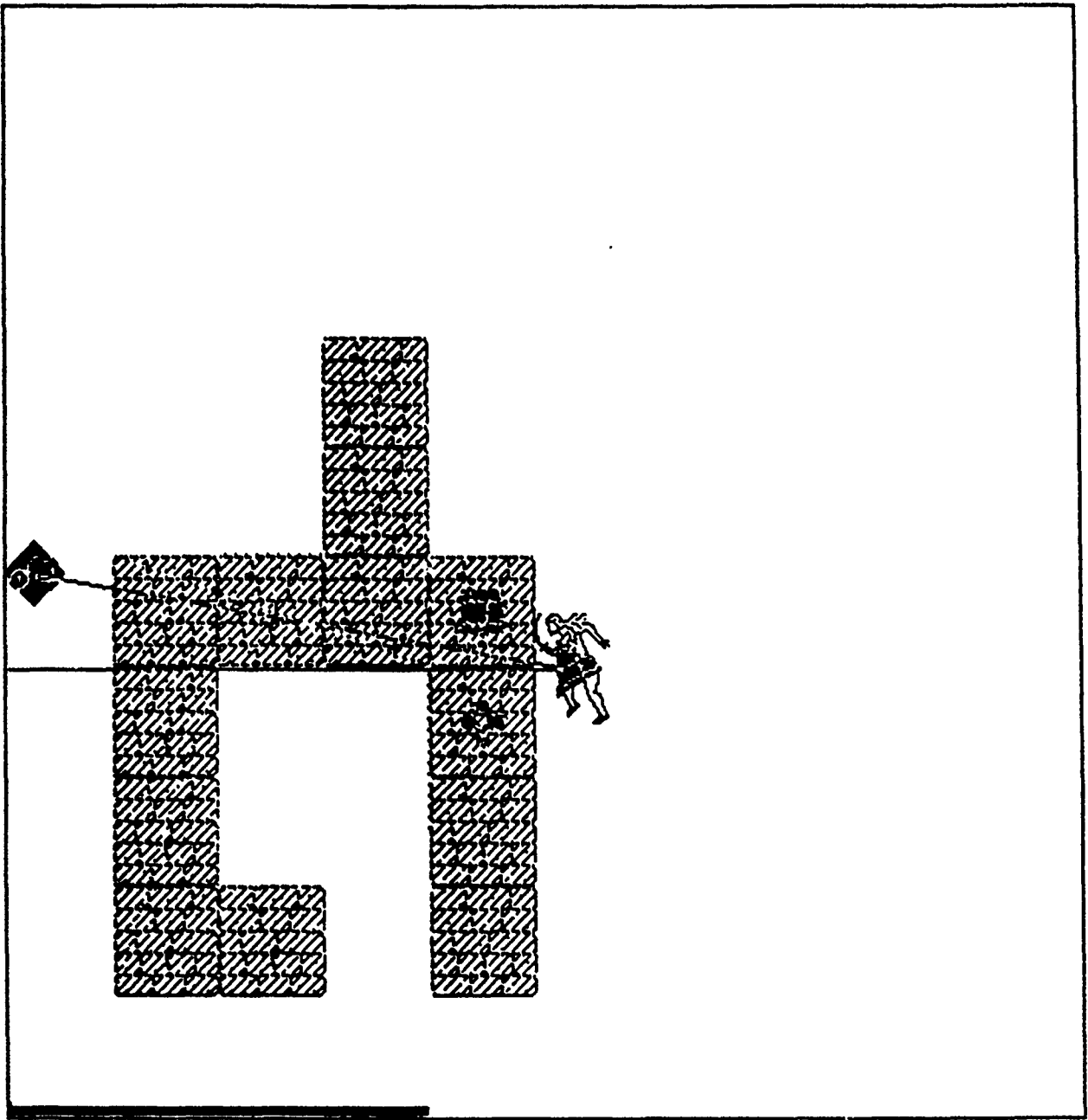


Figure 15: Activating the whole obstacle.

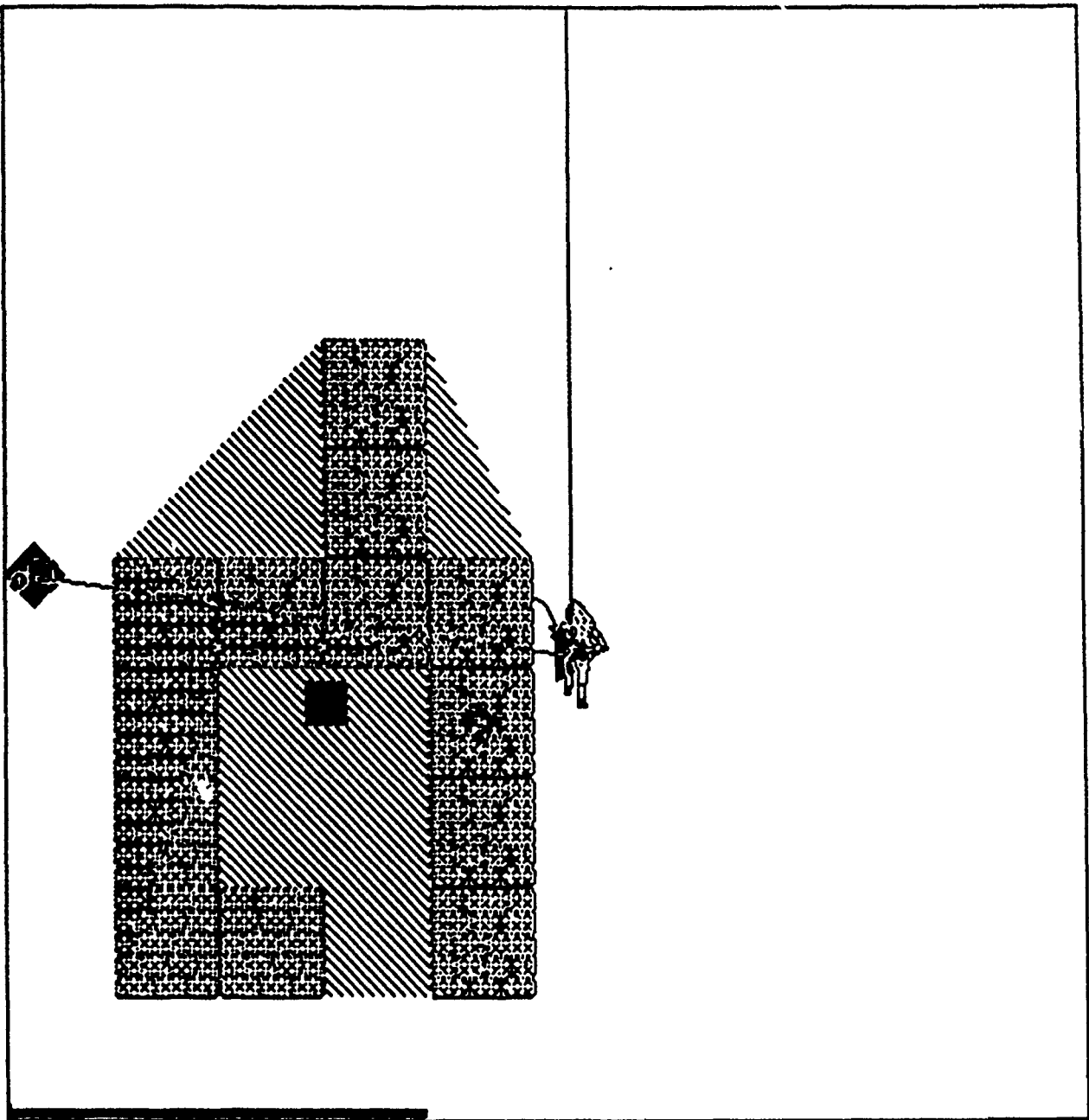


Fig 16: Marking the center of the convex hull of the obstacle to find the shorter way around. The shorter way is given by the sign of the angle between the destination, amazon, and obstacle markers: counterclockwise, in this case.

screen shows only part of a much larger underlying world.) Sonja determines whether the obstacle passes off screen by activating the portion it would pass around and determining whether that portion touches the edge of the screen. Specifically, Sonja uses `transect-activation!` to activate the portion of the obstacle that is on the appropriate side of the goal line (figure 17). Sonja then uses an operator called `activation-touches-screen-edge?` to determine if the obstacle runs off the screen in the direction it hopes to pass. This is the "inelegant" operator I mentioned in section 6.1; it is not obvious how to treat the edge of the screen given that SIVS does not truly implement early vision.

Now Sonja has a best guess as to which way to go around: the apparently shortest way unless it runs off the screen. To pass the obstacle, Sonja first sets the amazon's heading to the major component of the direction to the goal: left, in this case. This makes the heading ray pass through the obstacle, a condition Sonja senses with `ray-activated?`. Sonja then tries successively more indirect candidate directions by repeatedly setting the amazon's heading to be 45 degrees from its current heading, rotated in the direction opposite to that in which the amazon will pass around the obstacle, until the heading ray no longer intersects the activated region. (See figure 18.)

Eventually Sonja will have gone far enough that it can turn to head more directly towards the goal. Periodically it forces the amazon's heading 45 degrees closer to the goal (the opposite rotation from that used to find an initial valid heading). If the heading ray is still clear of the obstacle, Sonja tries again, rotating until it has gone too far and runs back into the obstacle (figure 19). Then Sonja rotates out one increment so that the ray is in the clear again. In figure 20 we see the amazon having gone far enough that a new heading is valid.

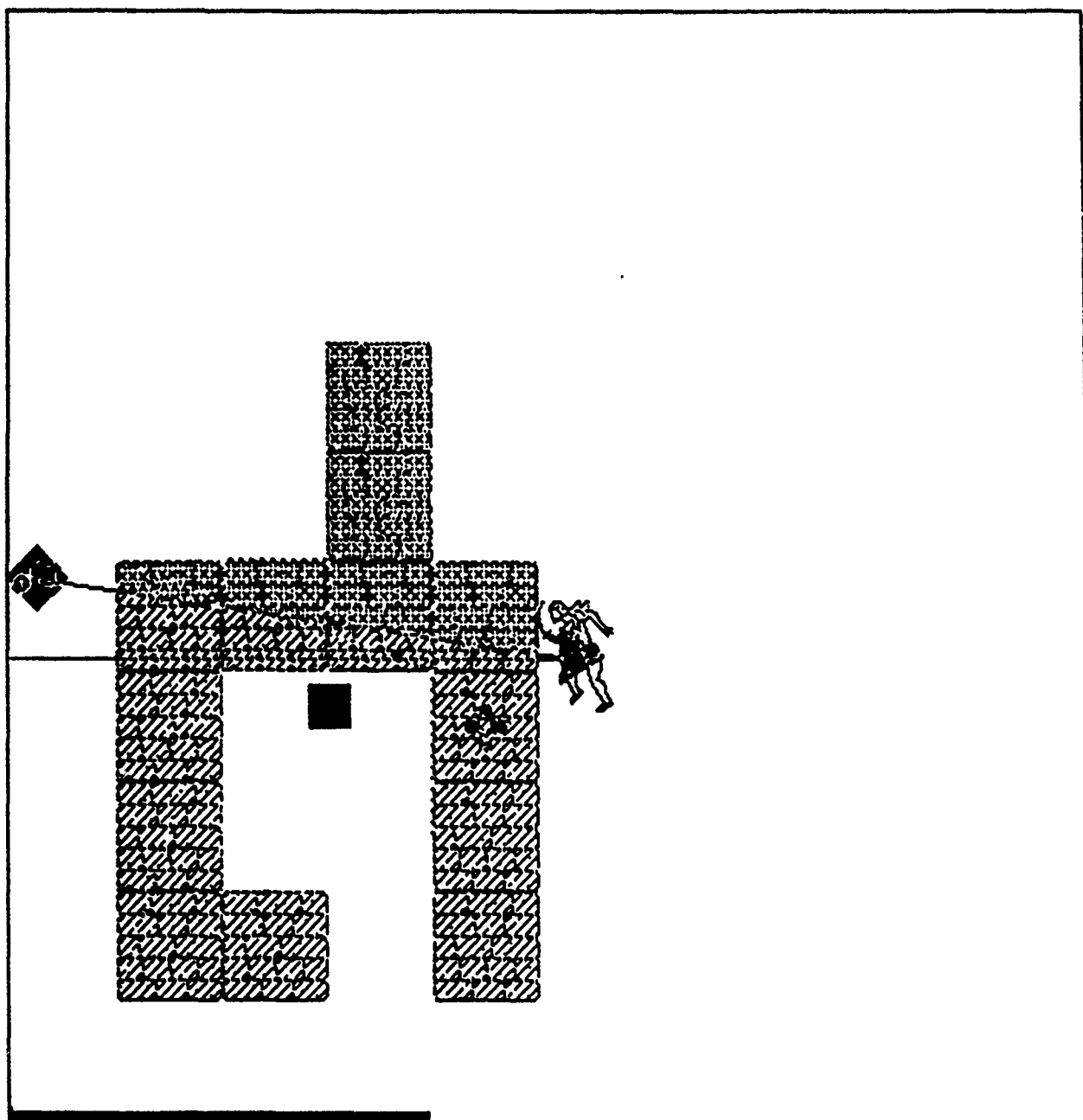


Figure 17: Activating the apparently shorter part of the obstacle. The upper portion is activated with both the original obstacle plane (diagonal increasing hatch pattern) and another (diagonal decreasing hatch pattern), yielding a lozenge pattern.

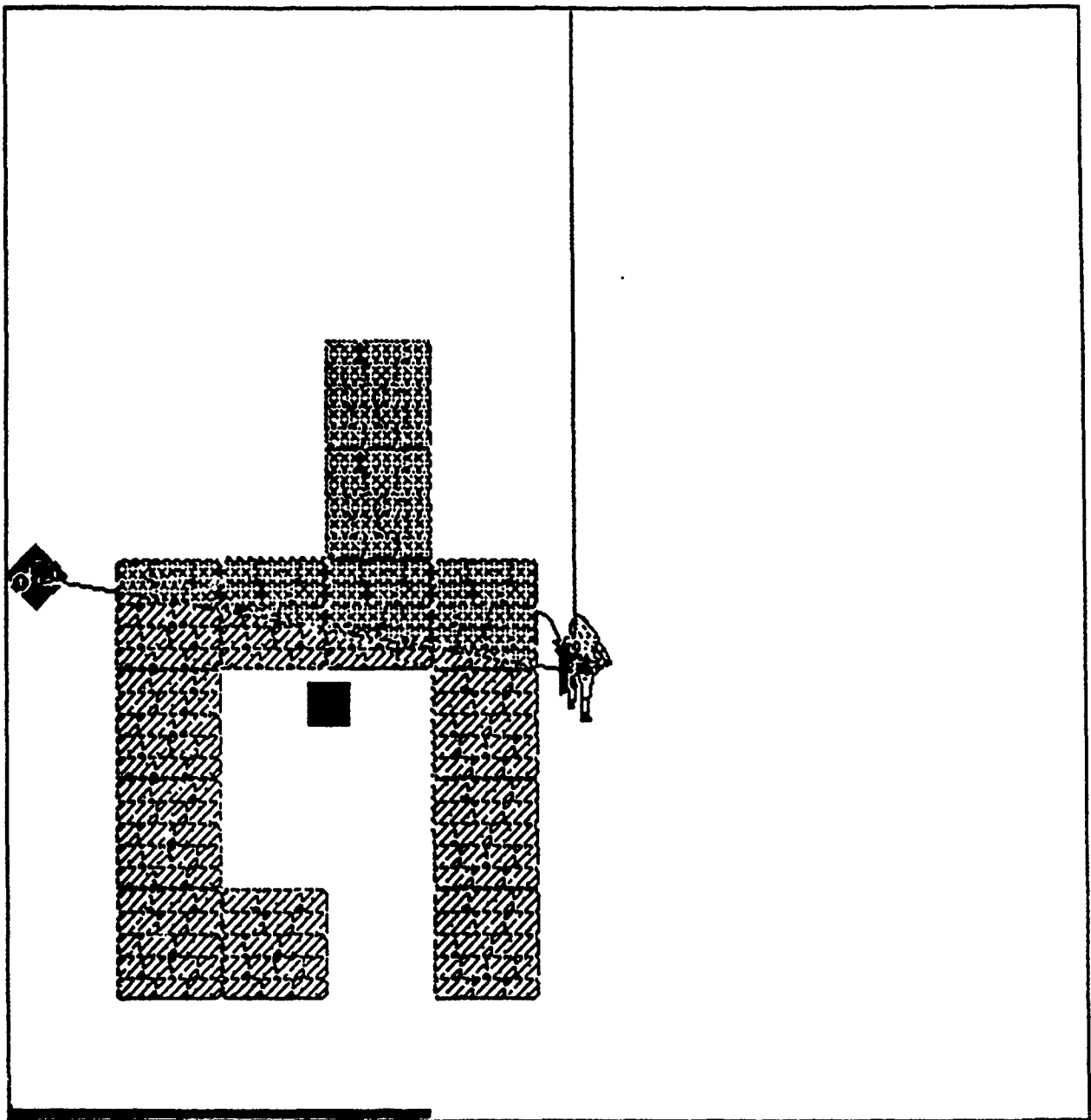


Figure 18: Searching for a heading. Sonja rotates the amazon, and the heading ray, progressively clockwise, until the ray no longer intersects the obstacle. In this case Sonja first tries heading left, finds that doing so causes the ray to intersect with the obstacle, and so tries heading diagonally up and left, which also fails. Finally it tries heading up, and succeeds.

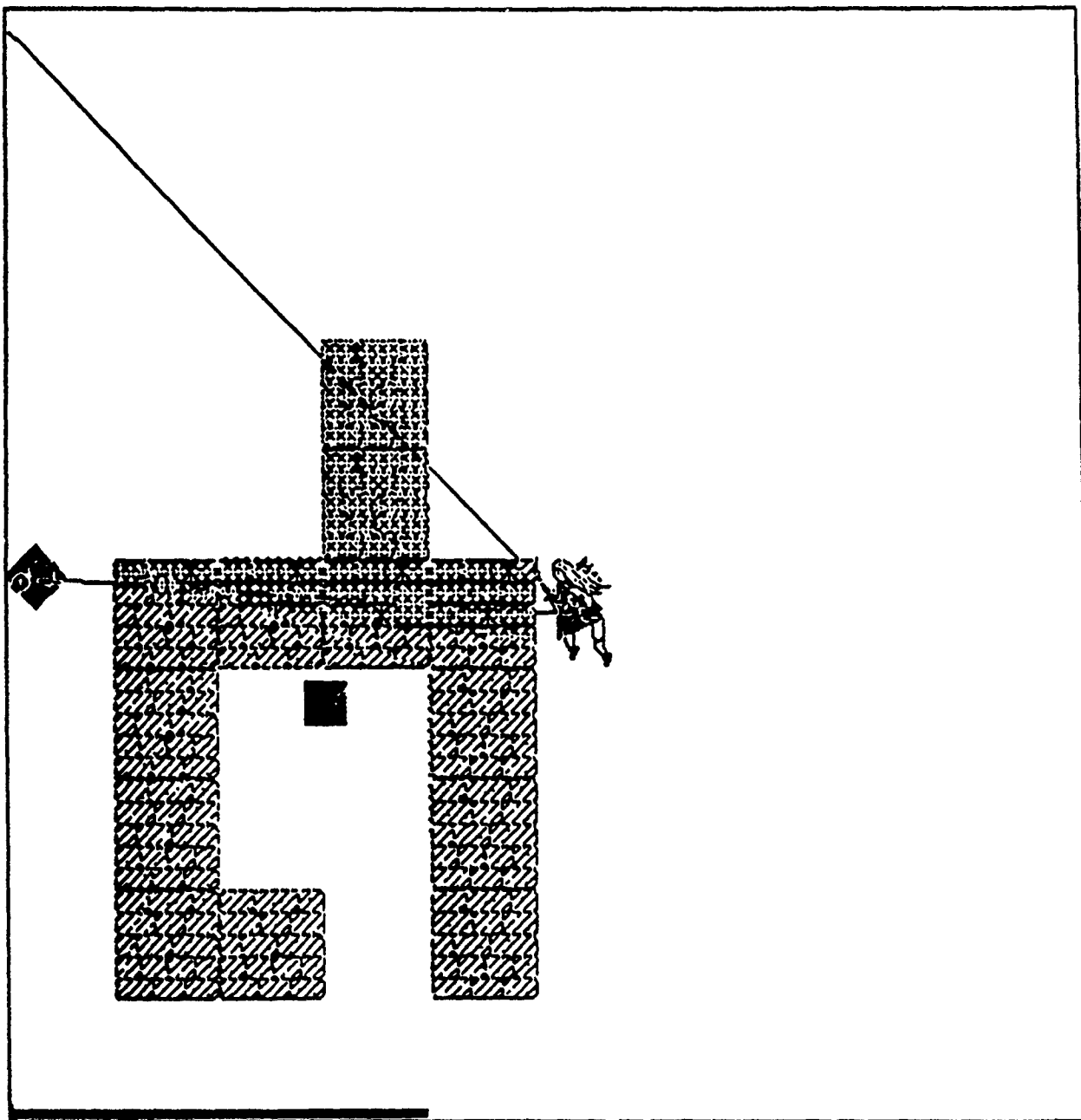


Figure 19: Rotating the heading ray back until it hits the obstacle.

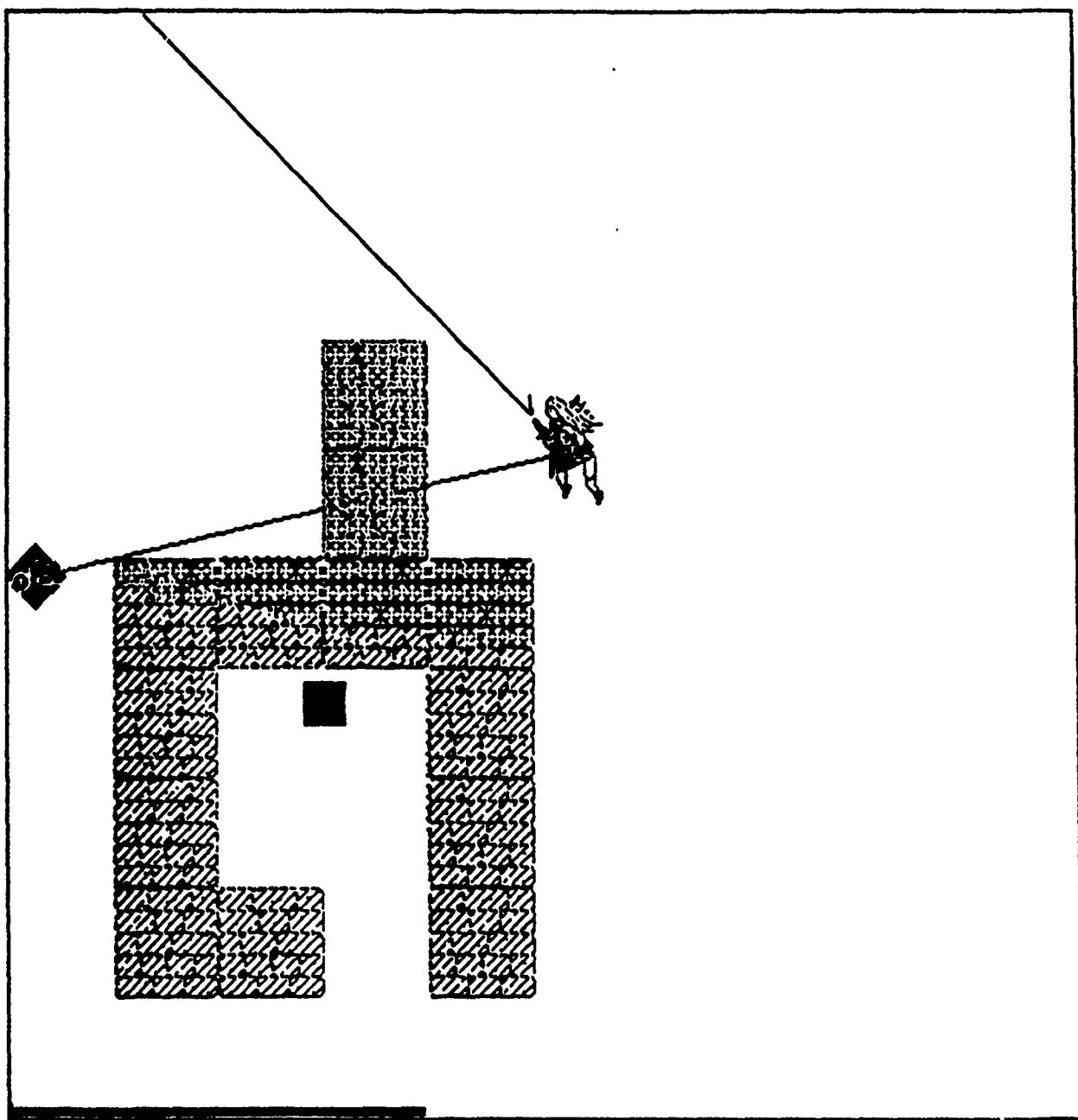


Figure 20: The amazon has gone far enough that a new heading will work.

8 Conclusions

SIVS is a first attempt at an integrated intermediate level vision architecture and so necessarily cuts some corners. Section 8.1 describes some of these cut corners. Section 8.2 evaluates SIVS according to the criteria of generality and usefulness posed in section 6.1. Section 8.3 summarizes the successes of the system. Though this exploratory investigation raises more questions than it answers, it demonstrates that familiar mechanisms such as visual search and attention can be made practically useful in a computational implementation.

8.1 Outstanding problems

The most pressing problems in both the model and implementation presented here are the lack of treatment of early vision and of object recognition.

SIVS bypasses all of the standard difficulties with early vision, such as noise and illumination variations. This raises the issue of whether this model of vision can be extended to domains in which early vision is harder. Unfortunately, other research on intermediate vision has similarly failed to address this issue. The relevant psychophysical studies use clean, evenly lit displays with highly discriminable stimuli. Ullman's visual routines paper uses as examples diagram interpretation tasks in which noise issues can be ignored, and this tradition has been continued by other researchers in the area. Noise sensitivity is a serious issue because it is the job of the visual operators to reduce noisy retinotopic arrays to compact encodings, carrying boolean values in many cases. So unlike the outputs of early vision, which vary continuously, the outputs of visual operators may be *very* wrong if they are not exactly right. Thus, they had better not be sensitive to noise. Is it possible to implement noise-insensitive operators that perform operations similar to those of SIVS? This is an open question on which the plausibility of the model rests. In current research I am connecting SIVS with a real-time early vision constructed by Nishihara [37, 38] in order to support a robot system.

SIVS does not address the hard issues in object recognition because Amazon is atypical in that objects can be categorized simply by combinations of early properties. The number of types of objects (icons) that can appear on the screen is small, and problems such as occlusion, the rotational instability of features, and non-rigid motion are not found in the domain.

Still, giving SIVS access to the icon datastructures does not fully finesse object recognition. What constitutes an object for the Amazon code and what constitutes an object for Sonja differ in some cases, so that Sonja has to do significant work to identify objects. Amazon represents walls in terms of uniform square chunks of wall-stuff, and represents only the positions of these chunks, not their connectivity relationships. Accordingly, as described in section 7, Sonja must use a connectivity operator to segment obstacles from the background.

Ullman originally proposed that visual routines are a preprocessing stage for shape recognition [60]. More recent work suggests that the two are parallel channels each connecting the early maps with the central system. It is tempting, in fact, to identify shape matching and the visual routines processor with the *temporal* and *parietal* visual processing streams, devoted respectively to object recognition and the assessment of spatial relationships, described by Ungerleider and Mishkin [61].

8.2 Evaluating the visual operators

This section evaluates Sonja's VRP according to the three "global" criteria posed in section 6.1.

- We want a "spanning" set: that is, a set of operators that together are sufficient for any task. Here "any task" may mean "any psychologically plausible task" or, for engineers, "any task in the class of domains of interest." Thus the set of visual operators, when combined into routines, are to form a finite means for the realization of an infinite collection of possible visual processes.

It is hard to formulate this criterion rigorously, but it is clear that SIVS does not satisfy it. I added operators to the set as needed for particular tasks. By the end of the implementation, I found I often had all the operators I needed to implement a new routine, but not always. This suggests that I may have been approaching an adequate set, but I'm sure that even in the one domain of playing Amazon continued system development would occasionally require new operators.

This raises the concern that there is no spanning set, or that it would be too large to implement with the amount of hardware found in the human brain. The visual routines model is only plausible if a relatively small spanning set is possible; since operators correspond to innate bits of hardware, there can be only as many as will fit in the brain. (Given that we know little about how intermediate vision is actually implemented, it is hard to say how many this would be. Hundreds might be feasible; millions probably wouldn't be.) Further research, for example cross-domain studies or formal analysis of the space of spatial reasoning tasks, is required to address this issue.

- A set of operators should not only make it possible to implement any visual task, it ought to make it *easy*. From an engineering standpoint, the VRP should present a nice programming system.

It is hard to evaluate SIVS on this score because I never had an opportunity to implement visual routines with a completed and debugged VRP. This largely negated the visual-system-as-a-programming-language metaphor I was trying to create. Only by the end of the implementation was the VRP relatively complete and reliable; by that time, implementing new routines was fairly straightforward. Further experience with the architecture is needed.

My feeling, however, is that the set of operators in SIVS is on the whole too *low-level*. Writing routines often seemed to require more work than it felt intuitively like it ought to. It required too much visual "bit diddling"; I wanted to be able to express things more abstractly. Such abstraction might be provided by higher-level operators. It might also, however, be provided by a "library" of general-purpose parameterized routines which would use low-level operators.

- A set of operators should also make it easy to *learn* new visual routines.

Lacking adequate theories of learning, it is hard to evaluate SIVS on this score. My thesis [7] suggests experiments that might elucidate the problem. It seems likely that learnability would be enhanced by the other criteria I have discussed. Learning is easier when your primitives span a space in which useful combinations are relatively dense [27].

8.3 Successes

This paper describes an implemented, integrated model of intermediate vision which addresses several fundamental but often neglected problems: selective application of visual processing to subsets of the image, search for regions of the image with task-relevant properties, and the computation of spatial relationships among parts of the image. These problems arise in most real visual tasks. SIVS's visual attention mechanism models psychophysical evidence. Its model of visual search is based on Treisman's theory [54, 55] and is the first implementation of visual search that models psychophysical results. SIVS's visual routines processor substantially extends Ullman's proposals [60], specifies plausible interfaces between visual routines and earlier and later processing, and is the first to apply visual routines to a natural task. All of these mechanisms are designed to be implementable in biological hardware. Sonja demonstrates the value of the mechanisms by using them to support visually-guided activity.

Because the biology of and computational constraints on intermediate vision are still poorly understood, many of the mechanisms proposed in this paper represent informed guesses. These proposals pose new open problems which can be the basis for experimental tests, many of which I have suggested.

Acknowledgements

This paper was improved by comments from Phil Agre, Leslie Kaelbling, Jeff Shrager, and Lambert Wixson.

References

- [1] Subutai Ahmad and Stephen Omohundro, "Equilateral Triangles: A Challenge for Connectionist Vision." *Proceedings of the 12th Annual Meeting of the Cognitive Science Society*, MIT, 1990.
- [2] C. H. Anderson and D. C. Van Essen, "Shifter circuits: A computational strategy for dynamic aspects of visual processing." *Proceedings of the National Academy of Sciences, USA*, Vol. 84, pp. 6297-6301, September 1987.
- [3] Dana H. Ballard, "Cortical connections and parallel processing: structure and function." *The Behavioral and Brain Sciences* 9 (1986) pp. 67-120.
- [4] Dana H. Ballard, *Eye Movements and Spatial Cognition*. University of Rochester Department of Computer Science TR 218, 1987.

- [5] Dana H. Ballard, "Reference Frames for Animate Vision." *IJCAI-89*, pp. 1635-1641.
- [6] David Chapman, "Penguins Can Make Cake." *AI Magazine*, Vol. 10, No. 4 (Winter 1989), pp. 45-50.
- [7] David Chapman, *Vision, Instruction, and Action*. MIT AI TR 1204, April 1990.
- [8] Francis Crick, "Function of the thalamic reticular complex: The searchlight hypothesis." *Proceedings of the National Academy of Science*, Vol. 81, pp. 4586-4590, July 1984.
- [9] Francis Crick and Christof Koch, "Towards a neurobiological theory of consciousness." Submitted, 1990.
- [10] Jon Driver and Gordon C. Baylis, "Movement and Visual Attention: The Spotlight Metaphor Breaks Down." *Journal of Experimental Psychology: Human Perception and Performance*. 15:3 (1989) pp. 448-456.
- [11] F. L. Engel, "Visual Conspicuity, Directed Attention, and Retinal Locus." *Vision Research* Vol. 11 (1971) pp. 563-576.
- [12] F. L. Engel, "Visual Conspicuity and Selective Background Interference in Eccentric Vision." *Vision Research* Vol. 14 (1974) pp. 459-471.
- [13] C. W. Eriksen and J. D. St. James, "Visual attention within and around the field of focal attention: a zoom lens model." *Perception and Psychophysics* 40 (1986) pp. 225-240.
- [14] Martha J. Farah, "Mechanisms of Imagery-Perception Interaction." *Journal of Experimental Psychology: Human Perception and Performance* 15:2 (1989), pp. 203-211.
- [15] M. J. Farah, J. L. Brunn, M. A. Wallace, and N. Madigan, "Structure of Objects in Central Vision Affects the Distribution of Visual Attention In Neglect." *Society for Neuroscience Abstracts* 15:1 (1989), p. 481.
- [16] Jerome A. Feldman, "Four frames suffice: A provisional model of vision and space." *The Behavioral and Brain Sciences* 8:2 (1985) pp. 265-313. With commentary from various authors.
- [17] Jerome A. Feldman and Dana Ballard, "Connectionist Models and their Properties." *Cognitive Science* 6 (1982) pp. 205-254.
- [18] Margaret M. Fleck, "Classifying Symmetry Sets." To appear in the *Proceedings of the British Machine Vision Conference*, 1990.
- [19] Kunihiro Fukushima, "A Neural Network Model for Selective Attention in Visual Pattern Recognition." *Biological Cybernetics* 55 (1986) pp. 5-15.
- [20] B. K. P. Horn, *Robot Vision*. MIT Press, Cambridge MA, 1986.

- [21] Pierre Jolicoeur, Shimon Ullman, and Marilyn Mackay, "Curve tracing: A possible basic operation in the perception of spatial relations." *Memory and Cognition* 1986, 14 (2), pp. 129-140.
- [22] Bela Julesz, "A brief outline of the texton theory of human vision." *Trends in NeuroScience*, February, 1984, pp. 41-45.
- [23] Raymond Klein, "Inhibitory tagging system facilitates visual search." *Nature* vol. 334 (4 August 1988) pp. 430-431.
- [24] Christof Koch and Shimon Ullman, "Selecting One Among the Many: A Simple Network Implementing Shifts in Selective Visual Attention." *Human Neurobiology* 4 (1985) pp. 219-227. Also published as MIT AI Memo 770/C. B. I. P. Paper 003, January, 1984.
- [25] Ben J. A. Kröse, "Local structure analyzers as determinants of preattentive pattern discrimination." *Biological Cybernetics* 55 (1987), pp. 289-298.
- [26] Ben J. A. Kröse and Bela Julesz, "The Control and Speed of Shifts of Attention." *Vision Research* Vol. 29 No. 11 (1989), pp. 1607-1619.
- [27] Douglas B. Lenat and John Seely Brown, "Why AM and Eurisko Appear to Work." *Artificial Intelligence* 23 (1984) pp. 269-294. An earlier, shorter version of this paper also appeared in *AAAI-83*, pp. 236-240.
- [28] Jim Mahoney, "Proposal for a system for spatial analysis of schematic drawings by visual routines." Unpublished M.S. thesis proposal, MIT AI Lab, June, 1985.
- [29] James V. Mahoney, *Image Chunking: Defining Spatial Building Blocks for Scene Analysis*. MIT AI Lab TR-980, August, 1987.
- [30] David Marr, *Vision*. W. H. Freeman and Company, San Francisco, 1982.
- [31] John H. R. Maunsell and William T. Newsome, "Visual Processing in Monkey Extrastriate Cortex." *Ann. Rev. Neurosci.* 1987 10 (1987) 363-401.
- [32] Jefferey Moran and Robert Desimone, "Selective attention gates visual processing in the extrastriate cortex." *Science* 229 (1985), pp. 782-784.
- [33] Michael C. Mozer, "A connectionist model of selective attention in visual perception." *Program of the Tenth Annual Conference of the Cognitive Science Society*, Montreal, 1988, pp. 195-201.
- [34] Michael C. Mozer, *The Perception of Multiple Objects: A Connectionist Approach*. MIT Press, Cambridge, MA, forthcoming.
- [35] K. Nakayama and G. H. Silverman, "Serial and parallel processing of visual feature conjunctions." *Nature* 320 (1986), pp. 264-265.

- [36] Ken Nakayama and Manfred Mackeben, "Sustained and Transient Components of Focal Visual Attention." *Vision Research* 29:11 (1989), pp. 1631—1647.
- [37] H. K. Nishihara, "Practical Real-Time Imaging Stereo Matcher," *Optical Engineering* 23, 5, 536-545, Sept.-Oct. 1984. Also in *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, edited by M. A. Fischler and O. Firschein, Morgan Kaufmann, Los Altos, 1987.
- [38] H. Keith Nishihara, "RTVS-3 Real-Time Binocular Stereo and Optical Flow Measurement System System Description." Teleos Research internal document.
- [39] D. I. Perrett, E. T. Rolls, and W. Cann, "Visual Neurones Responsive to Faces in the Monkey Temporal Cortex." *Exp. Brain Res.* (1982) 47:329-342.
- [40] Michael I. Posner, Charles R. R. Snyder, and Brian J. Davidson, "Attention and Detection of Signals." *Journal of Experimental Psychology: General*, 1980, Vol. 109, No. 2, pp. 160-174.
- [41] M. I. Posner, Y. Cohen and R. D. Rafal, "Neural systems control of spatial orienting." *Philosophical Transactions of the Royal Society of London, Series B* 298 (1982) pp. 187-198.
- [42] Mary C. Potter, "Representational Buffers: The Eye-Mind Hypothesis in Picture Perception, Reading, and Visual Search." Chapter 24 in *Eye Movements in Reading: Perceptual and Language Processes*. Academic Press, New York, 1983.
- [43] Zenon Pylyshyn, "The Role of Location Indexes in Spatial Perception: A Sketch of the FINST Spatial Index Model." *Cognition*, 32:1 (June, 1989), pp. 65-96.
- [44] Zenon W. Pylyshyn and R. Storm, "Tracking multiple independent targets: Evidence for a parallel tracking mechanism." *Spatial Vision* 3:1-19.
- [45] Marc H. J. Romanycia, "The Design and Control of Visual Routines for the Computation of Simple Geometric Properties and Relations." University of British Columbia Department of Computer Science Technical Report 87-34, 1987.
- [46] H. Sakata, H. Shibutani, and K. Kawano, "Functional properties of visual tracking neurons in posterior parietal association cortex of the monkey." *Journal of Neurophysiology* 49 (1983) pp. 1364-1380.
- [47] A. Shafir, "Fast region coloring and computation of inside/outside relations," M. Sc. Thesis, Department of Applied Mathematics, Feinberg Graduate School, Weizmann Institute of Science, Rehovot, Israel.
- [48] Mary L. Shaw, "A capacity allocation model for reaction time." *Journal of Experimental Psychology: Human Perception and Performance*, 4 (1978), pp. 586-598.

- [49] M. L. Shaw and P. Shaw, "Optimal allocation of cognitive resources to spatial locations." *Journal of Experimental Psychology: Human Perception and Performance*, 3 (1977), pp. 201-211.
- [50] J. Shrager, D. Klahr, and W. G. Chase, "Segmentation and Quantification of Random Dot Patterns" *Proceedings of the 29rd Annual Meeting of the Psychonomics Society*, 1982.
- [51] Gary W. Strong and Bruce A. Whitehead, "A solution to the tag-assignment problem for neural networks." *Behavioral and Brain Sciences* (1989) 12, pp. 381-433.
- [52] William B. Thompson and Ting-Chuen Pong, "Detecting Moving Objects." *Proceedings of the First International Conference on Computer Vision*, 1987, pp. 201-208.
- [53] Anne Treisman, "Perceptual Grouping and Attention in Visual Search for Features and Objects." *Journal of Experimental Psychology: Human Perception and Performance*, Vol. 8 No. 2 (1982) pp. 194-214.
- [54] Anne M. Treisman and Garry Gelade, "A Feature-Integration Theory of Attention." *Cognitive Psychology* 12 (1980), pp. 97-136.
- [55] Anne Treisman and Stephen Gormican, "Feature Analysis in Early Vision: Evidence From Search Asymmetries." *Psychological Review* Vol. 95 (1988), No. 1, pp. 15-48.
- [56] Anne Treisman and Janet Souther, "Search Asymmetry: A Diagnostic for Preattentive Processing of Separable Features." *Journal of Experimental Psychology: General*, Vol. 114, No. 3 (September 1985), pp. 285-310.
- [57] Yehoshua Tsal, "Do Illusory Conjunctions Support the Feature Integration Theory? A Critical Review of Theory and Findings." *Journal of Experimental Psychology: Human Perception and Performance* 15:2 (1989), pp. 394-400.
- [58] John K. Tsotsos, "Analyzing vision at the complexity level." To appear, *Behavioral and Brain Sciences*.
- [59] Shimon Ullman, "Filling in the Gaps: The Shape of Subjective Contours and a Model for their Generation." *Biological Cybernetics* 25:1 (1976), pp. 1-6.
- [60] Shimon Ullman, "Visual Routines." *Cognition* 18 (1984), pp. 97-159. Also published as MIT A.I. Memo 723, June 1983.
- [61] L. G. Ungerleider and M. Mishkin, "Two cortical visual systems." In D. J. Ingle, M. A. Goodale, and R. J. W. Mansfield, *Analysis of Visual Behavior*, MIT Press, Cambridge, MA, 1982.
- [62] Jeremy M. Wolfe, Kyle R. Cave, and Susan L. Franzel, "Guided Search: An Alternative to the Feature Integration Model for Visual Search." *Journal of Experimental Psychology: Human Perception and Performance* 15:3 (1989), pp. 419-433.

- [63] S. M. Zeki, "The functional organization of projections from striate to prestriate visual cortex in the rhesus monkey." *Cold Spring Harbor Symposium on Quantitative Biology* 40 (1975), pp. 591-600.
- [64] S. M. Zeki, "Uniformity and Diversity of Structure and Function in Rhesus Monkey Prestriate Visual Cortex." *Journal of Physiology* 277 (1978), pp. 273-290.

F Cooperative Robot Demonstration: Working Document

Cooperative Robot Demonstration: Working Document

Leslie Pack Kaelbling
Neil D. Hunt
Stanley J. Rosenschein
H. Keith Nishihara
Nathan J. Wilson
Laura E. Wasylenki
Jeffrey R. Kerr

October 6, 1989

Chapter 1

Motivation and Plan

1.1 Outline

We would like to produce a demonstration of robotics capabilities illustrating approaches and techniques in the areas of real time perception, reactive control and planning, and manipulation. An important characteristic of our approach is to make all the elements of the demo as general as possible, so that we can extend the system in many different directions to enable additional demonstration capabilities.

The goal of this plan is to formalize our approach, and to concentrate our efforts so that a large group of people might contribute fully to a significant combined effort.

1.1.1 Philosophy

Our aim is to create a set of generalized methods at each level in the architecture which we hope will lead to a cross-product of capabilities. The addition of a new capability at any level *multiplies* the performance of the whole system rather than simply *adding* to it.

We intend to develop a system which is clearly programmed in a generic way, in which the subcomponents are each applicable in a wide range of situations, so that the whole system can respond to situations far outside those few which could have been enumerated by the designers of the system and of each component. We hope to demonstrate incremental progress down a well defined path toward such richness of performance.

1.1.2 Methodology

We will tackle this project in a series of stages or slices, each of which will have two main phases:

1. Define an incremental level of performance which can be achieved in a period of about a month. Define the architecture, interfaces and communications languages of the complete system. Determine the additional capabilities which will be required at each level.
2. Create a skeleton system in which each module exists in some nominal fashion. The skeleton system should allow developments in each separate module to be at

least partially tested in its environment. After the first stage, this slice will involve something like implementing the extended interface definitions and recompiling.

3. Extend the capabilities of each module in parallel, to include those requirements identified at the start of each stage.
4. Integrate all the modules, test and evaluate the system.

1.1.3 Robot games

A proposed sequence of performance goals will permit the system to play 'robot games' such as those listed below.

1. Toggle (pick up, put down; hold this, hold that).
2. Hide and Seek.
3. Passing back and forth. (Human hand)
4. Tag?
5. Catch.
6. Pour me a drink.
7. Simon says (user specified goals in some language).

It should be emphasized that we hope to build a system whose capabilities cannot be enumerated, where the compositional structure of the goals provides great richness. The games listed above are simply intended to motivate and illustrate initial subsets of the possibilities.

1.2 Architecture

We conceive of the system as having three components: next state computation, internal state, and action control, as illustrated in Figure 1.1. The system operates in a regular, clocked mode, in which the current state vector is updated based upon new inputs and the previous state.

The three components can be divided into a number of subcomponents. We identify three major components of the internal state:

Recognition state Recognition of objects and characteristics of the world must take place in stages over time, with successive observations accumulating evidence toward recognition. This accumulating evidence is part of the internal state of the system.

Database The system maintains its current picture of the world in the form of an objects and properties database.

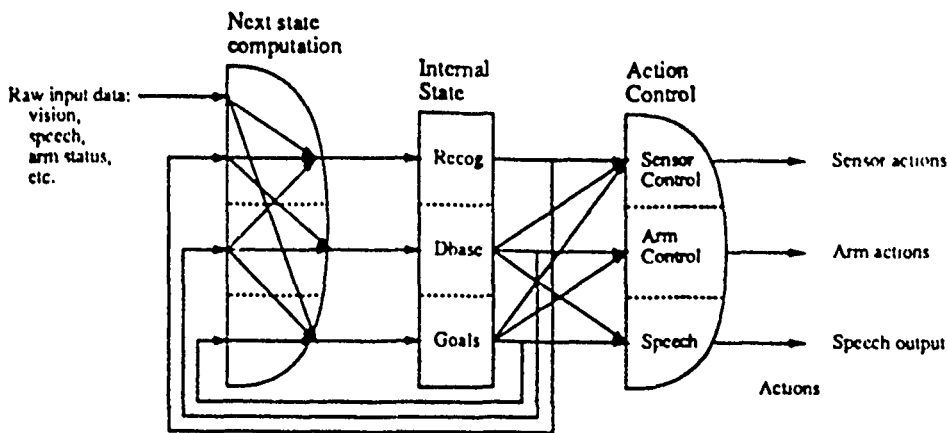


Figure 1.1: Proposed demo system architecture.

Goals The current goals (volition) of the system is encoded in another piece of the internal state.

Each of these subdivisions of the internal state vector is updated based upon different mixtures of previous state and new inputs. Thus the next state computation may also be divided into the same three sub parts.

Recognition The recognition component is responsible for accumulating evidence about properties of the world in the recognition state; to do this, it considers raw input from the world, the properties of other objects in the world (obtained from the database state), and previous evidence collected in the recognition state. In addition, manipulator status is recognized and used to update properties of the manipulator objects in the database.

Database The database of world properties is maintained by degradation of the information over time. The main input to the next state computation for the database is the previous state of the database, which contains all the data about velocities, accelerations, etc. for each object. In addition, the recognition subsystem can propose new objects for the database, or may be able to retract previous assertions made to the database. Thus additional input is obtained from the recognition state.

Goals The goals of the system are derived from speech input data, and are maintained over time by copying the previous state.

Based upon the current input state, the system can take various actions to satisfy its goals. The actions can also be broken down into several parts:

Sensor control Perception of the world is an active process, and demands that the sensors generating the raw perceptual data are pointed and controlled to acquire the data most useful for recognition of conditions in the world. The sensor control actions are dependent upon the current recognition state, and also upon the current objects in the world. They are also dependent upon the current goals of the system.

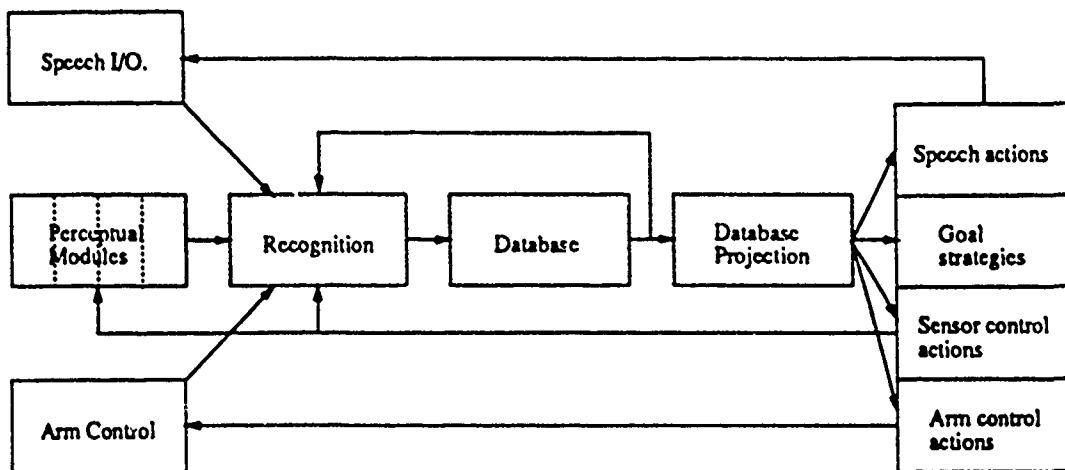


Figure 1.2: Proposed demo system architecture.

Arm control The manipulators are the principal means by which the system interacts with the world to achieve its goals. The arm control outputs depend upon the objects currently in the world database, and also the goals of the system.

Speech output Speech (or other textual output) is important for answering queries, and for prompting or asking for assistance or clarification from the user of the system. The speech output is based upon the current goals and the state of the objects in the world database.

While the model of the system presented above is conceptually simple and elegant, it is more convenient to break it down into horizontal slices to enable different pieces to be constructed in parallel as separate modules. Figure 1.2 shows a modular decomposition of the architecture described above.

The relationship between the parts of the two figures is described below:

Perceptual modules The raw input data requires a great deal of specialized processing before it is suitable as updates to the recognition state; The perceptual modules of Figure 1.2 encapsulate such processing, and are viewed as a set of tools which can be pointed and controlled in order to ask specific questions about the state of the world. They receive inputs from the sensor-control action module in addition to raw world data, and generate outputs to the recognition system.

Recognition The recognition module of Figure 1.2 groups together the recognition computations, and also the recognition internal state. The third part of the recognition

slice is the sensor control section.

Database The database component includes the database state, and the update rules. Inputs are obtained from the recognition (both world properties and textual inputs).

Database projection The action block of Figure 1.1 must test many different conditions about the world represented in the database. In order to ensure that the development of each of the parts of the actions block is well structured, the computation of predicates representing these world conditions has been abstracted into a block called database projection (because it projects the facts in the database into a set of conditions and properties of direct interest for determining actions to perform). A central task is to determine a language for specifying the conditions that will be implemented in database projection, that will motivate the type of recognition and perception capabilities which will be required, and that will control the type of actions which can be implemented.

Goal strategies and action control The actions block remains grouped together, as the different actions which can be performed must be mediated by some common system so that common resources (manipulators, time, attention) can be allocated according to priorities established by the goals, and by a set of *top level goal achievement strategies*. The goal strategies will have available sets of action strategies furnished as *speech actions*, *sensor control actions*, and *manipulation actions*.

Arm control The arm control block translates the action commands from the actions block into low-level manipulator commands, and feeds status information and force data back into the recognition module.

Speech control The speech control block translates the speech commands from the actions block into low-level speech commands. It also interprets speech input and feeds the data to recognition.

The next sections discuss the requirements for each of these modules in more detail.

1.2.1 Perception

A number of perception modalities are anticipated, including vision, touch/force, and sound.

The visual perception capability is viewed as a set of modules making reliable, robust, real-time measurements of physical parameters of the world. They are to be viewed as tools, which can be controlled and queried by the higher level processes.

Touch and force feedback is afforded by the robot manipulators available and will be used as an additional capability for determining details of the world.

Speech input and output (initially simulated using a keyboard and screen) will be used for goal acquisition and for the system to interact with users by outputting status or requests for the user.

1.2.2 Recognition

The recognition component will take as input the results of queries made of the low-level perceptual machinery and force-feedback data and will transduce that into high-level statements about objects and their relations. The objects that we would like to recognize (at least for the initial stages of the project) are: table, robot arm and hand, human arm and hand, coffee cup, soda can, ping-pong ball, and freezer container. The recognition component will have its own internal state and will be responsible for managing the uncertainty connected with intermediate stages of recognition. When it passes information out to the next component, it will be taken to be true.

The recognition process also has an action component. It will be important to direct the low-level perceptual sensors to parts of the world that will be of most use to the recognition component. In a similar vein, the recognition actions will be triggered by high-level goals, which will tell the robot to look for ping-pong balls, or track the human hand, or search for a flat surface on which to place some object. The recognition-action component may also direct the arm in order to use the force-feedback to find the location of an object. Goals of information will have to be prioritized with other goals in order to avoid conflicts.

The recognition process may also use information from the database to guide its perceptual activities and inferences. An interface must be carefully specified in order to shield this component from the implementation details of the database.

The recognition component can be tested, at the early stages, with no other components, by giving the agent top-level goals of information and simply tracing the outputs of the recognition module.

Individual steps in building the recognition component are hard to identify. A graded set of abilities can be described, however, as follows:

1. Static object alone against contrasting background; no occluding or even distracting objects.
2. Static object in scene with others, but no occlusion.
3. Slowly-moving object with no occlusion or confusion.
4. Slowly-moving object with other objects but no occlusion.
5. Static objects that occlude one another.
6. Moving objects that occlude one another.

1.2.3 Database

The database component will be responsible for storing descriptions of objects found by the recognition component. It will store information about a finite number of objects and their relations with one another in very general terms. This information will include a tag that describes the type of the object and bounds on the locations of objects with respect to one another as well as their relative velocities. The information will come from the recognition component.

This component can probably be developed and tested initially by using synthetic data. An important early decision is whether to implement it in C or in Rex. If we do it in C it will require a fairly large amount of system hacking to get the interfaces to work right; doing it in Rex entails a fairly large performance penalty due to the inefficiency of indexing, but would make it easy to integrate with the rest of the system.

The database must perform the following operations:

Merging: If two database entries can be shown to be the same physical object (perhaps because they occupy the same space), then their parameters should be intersected and one of the instances deleted.

Propagation: If it is possible to deduce something about the relation between objects i and k from the relations between objects i and j and objects j and k , then the relation between objects i and k should be strengthened accordingly. It is difficult to bound the number of propagation steps required to get the database into steady-state. In practice, a certain number of propagations will be performed each tick, leaving some relational information incompletely localized.

Degradation: As time passes, the information that the database has about a particular object will degrade. The database must degrade its information every tick, based on what it knows about individual objects. For instance, it might know that a coffee cup with no arms near it will stay where it is from one tick to the next, or that the robot arm can only move at a certain maximum velocity so that it must be within a certain distance of where it was last tick.

Inconsistency reduction: Although the recognition component strives to come to true conclusions, sometimes it will err. The database may notice such errors in either the merging or the propagation phase. When it tries to intersect two sets of properties and relations for an object and finds a contradiction, it must remove all of the objects involved. For instance, if it decides that two entries must be the same because they occupy the same space, but that they are different colors, it will throw them both out.

Pruning: There will be only enough room in the database for a finite number of objects (in order to keep the update time bounded). When a new object is found by the recognition component and the database is full, it must decide which object to throw away. This may be based on the recency or specificity of information about the object. It might be more reasonable to make it depend to some degree on the agent's current goals; this would require feeding goal information back to the database component.

1.2.4 Manipulation

The manipulation component will consist of a set of low-level manipulation strategies, written in Gapps, that achieve or maintain particular goals. These strategies are expected to make use of conditions that can be derived from information in the database as well as (perhaps) direct use of force-feedback data. The implementors of the manipulation

component should tell the database implementors if they would like to have the force information directly or channeled through the database.

Many of these abilities can be debugged initially by fixing the locations of the objects and using force information directly (unless we want or need to do visual servoing; if this is the case, we may want to work on data directly from the recognition component or from the database). We may want to invent more abilities that show off force control or other novel capabilities of the robot and its programmers.

Below is a list of possible manipulation abilities. Although they are listed individually, it would be best if the entire set of abilities could be generated from a few major kinds of manipulations and different sets of parameters.

Desired manipulation abilities include:

1. Pick up ping-pong ball.
2. Pick up cup (from different orientations).
3. Pick up soda can.
4. Pick up freezer container (from different orientations).
5. Open soda can (if possible).
6. Pour liquid from one vessel to another (soda can to cup is a good starting case).
7. Place picked-up objects back on table.
8. Put ping-pong ball inside cup or freezer container.
9. Put freezer container or cup over top of ping-pong ball.
10. Put object in human's hand.
11. Hold objects for human to take, noticing when he has a hold of it and letting go.
12. Take objects from human.

1.2.5 High-level Action

The high-level action component will initially consist of a set of strategies for playing interactive "games" with a human user. A more sophisticated version will have the user describe (in natural-ish language) what game he wants to play.

Some possible games are:

Pick-up/Put-down: This game can be played with any object; the robot is to pick up the object if it is on the table and put it down if it is not. Can be generalized to toggle between any two user-specified conditions. The conditions can be finite conjunctions (or priors) of ach's and maint's. Need way of specifying to the robot that it should stop the current game and start another one.

Catch: In the simple case, give an object to the human, then take it from him. In the more complex case, play real catch by throwing the object. Not sure we have enough precision in when the gripper opens and closes to do this. Could catch the ball in a cup and "throw" it by dropping it.

Hide and Seek: Tell the robot to find a particular object. Robot indicates the object (somehow) when it is found. May entail physical actions like picking up an overturned coffee cup to see if there is a ping-pong ball under it.

Tag: Dangerous. Robot tries to tag human hand, then human tries to tag robot hand while robot tries to get away.

Simon Says: All of these games can be expressed as fairly simple Gapps goals. In the long run, implement run-time goal-reduction and all the Gapps goal operators at run time with a fixed bound on total goal length. With a suitable set of primitive conditions, there could be a wide variety of interesting robot games. This ability can be approximated in the shorter term by having a set of high-level goal types and allowing the user to give the agent simple conjunctions of parametrized instances of the goal types. An example would be: (ach, maint) (on, in) <referring-expr> <referring-expr>. Syntactic sugar could be used to make the language more natural.

1.2.6 Database projection

In the course of writing the high-level action strategies, the programmer will need to test a number of conditions. These conditions will not be directly available in the database, but should be projectable from the information contained in the database. In addition, there will be a set of standard world conditions that the human can use to communicate with the robot. The task of high-level perception is to implement the functions that map the database into these conditions. They will be indexical-functional conditions like the-location-of-the-cup-containing-the-ping-pong-ball and is-there-a-unique-cup-containing-a-ping-pong-ball.

It might be reasonable to write these functions in Ruler, although, because these are pure functions, we wouldn't need the state-update facilities. The standard world conditions can be specified ahead of time, but much of the specification of this task will depend on the specific conditions that need to be tested in order to carry out particular strategies. We will use a simple compositional language to specify these conditions rather than using atomic names.

1.3 Interfaces and Implementations

1.3.1 Machines and languages

Perceptual modules: Initially, stereo and motion modules will be implemented on Natasha (symbolics lisp machine), and shape and other capabilities will be implemented on Boris (Decstation 3100). Since there is no frame buffer capability on Boris.

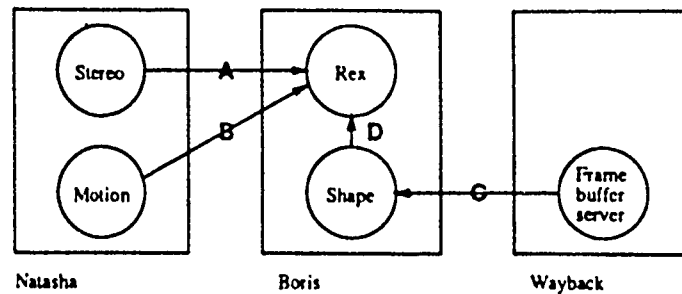


Figure 1.3: Processes and machines.

Wayback (Sun 2/120) will be hooked up as a frame buffer server passing portions of image to Boris across the ethernet.

Recognition: Recognition will be implemented in Rex, to run as part of a monolithic process (incorporating the database, database projection, and action rules) on Boris.

Database: The database will probably be implemented in C, but will be linked into the Rex module running on Boris. Nathan has described a technique for generating a shell database by defining inputs and outputs and a minimal Rex program, and causing the Rex compiler to generate the shell which will then be filled in.

Database projection: The database projection rules will be written in Rex and incorporated into the main process.

Actions: Each of the actions modules will be written as a set of Gapps rules, which will then be integrated by the Gapps compiler into the main process running on Boris.

1.3.2 Synchronization issues

We have decided to maintain the asynchronous behaviour between the different processes of the system, as it offers significant simplifications in communications protocols, and offers possibilities for running the system in a fault tolerant manner.

However, where multiple processes are running on the same machine (Boris), there was a question as to whether the processes should communicate directly to avoid pathological timesharing behaviour, or whether indirect synchronization or no synchronization would suffice.

No synchronization: The round robin type timesharing operating system would ensure that each process obtained its share of the resources over time, but it was questioned whether the granularity of such sharing would be satisfactory for maintaining real-time performance.

Indirect synchronization: One possibility would be simply to have the cooperating processes simply relinquish control of the CPU at suitable intervals by sleeping for a few milliseconds, enabling the other process to resume action. Suitable points would be at the beginning or end of each Rex tick, and between working on different perception queries.

Direct synchronization: Apart from hacking the scheduler (or obtaining a real time operating system with the necessary facilities already present), other more plausible solutions include synchronizing the two processes by means of pipes, signals, or other IPC mechanisms.

Desirable properties of the mechanism chosen include satisfactory performance, and maintenance of modularity between the different processes. It would be good if the processes could be moved to alternative processors without changing the code too much.

The hope is that with Rex sleeping after each tick up to the declared tick time, the scheduler will be able to give the perception processes sufficient compute time for the system to work. If this turns out not to work, we can either switch back to having Rex call the perception processes explicitly or have some synchronizing message passing between the competing processes.

1.3.3 Interfaces

Most of the interfaces between the modules of Figure 1.2 will be internal data structures within Rex. The definition phase will determine the nature of the data exchanged across such interfaces.

A number of external interfaces between processes and machines also need to be designed.

1.3.3.1 Recognition and Perception (interfaces A, B, D)

This section explains how we expect code written in Rex to control and get information from a set of perceptual processes each capable of performing some parametrized calculation on some input unavailable to Rex. It is designed to handle situations in which many questions can be answered in one tick, as well as situations in which it takes many ticks to answer a single question.

The general model is that each tick Rex will ask up to a fixed number of questions in a prioritized order. Each tick it will also receive up to a fixed number of answers to questions that it asked sometime in the past.

The questions are sent to the Rex Execution Environment, hereafter RexEx, in the usual way. (Note: In the current implementation the Rex code is actually called directly by RexEx and hence is just a part of the RexEx process.)

RexEx takes each question and tags it with the tick number in which the question was asked. It then sends the questions, in priority order, to whichever process it considers appropriate, as a packet through a socket. Using sockets allows us to have either processes running on separate machines connected to the machine running RexEx, or as separate processes running on the same machine. Note that it is also possible to have a single process

that answers the questions for more than one socket. Initially the different processes will answer very different types of questions so they will probably be dispatched based simply on a type field in the question from Rex. In the future, if we ever have a system where more than one perceptual process is capable of answering the same question, a more complex dispatching system could be devised. After dispatching all the questions asked by Rex, RexEx sends a null question packet to all the perception processes that were not asked any question. As with the other question packets, the null question packets include the current tick number.

Each perception process maintains a queue of questions. When a perception process has an empty queue it blocks, waiting for new questions to arrive on its socket from RexEx. When a question is read by a perception process, any questions in its queue from earlier ticks are pitched and the new question is put on its queue. Once the socket is empty the process takes off the first question in the queue and starts working on it. When it is done it sends the answer back to RexEx, looks for new questions and starts over. Note that since null packets are labelled with their tick numbers, they cause the queue to be purged.

After sending out the questions, RexEx looks for any answers that might have arrived from the perception processes. If more answers arrive on a given tick than can be sent to Rex, the oldest answers are discarded. A more complex selection scheme may need to be developed if it turns out that we are discarding a lot of answers.

We considered a number of more complex schemes for handling overruns of questions going to the perception processes or answers coming back to Rex. The systems we considered involved giving each question a unique ID number and/or a priority number over some small range.

1.3.3.2 Framebuffer server to Perception (interface C)

This will be a two way ethernet protocol, in which perception indicates the subimage desired from which frame buffer and at what resolution. The existing UDP packet protocol could be used, subject to packet size constraints. Alternatively some higher level protocol will be used. This will be decided in the next phase.

Chapter 2

Level 1 Demonstration

2.1 Task Definition

This section gives an external specification of the desired abilities of the level 1 demonstration. Note that it is not sufficient to simply achieve these abilities; the code should be structured according to the general philosophy and methodology of this project and be amenable to extension into the next phases of the demonstration.

The system should be able to satisfy goals of the form

$$\{ach \mid maint\}(\langle verb \rangle \langle noun \rangle \langle noun \rangle)$$

in which $\langle verb \rangle$ ranges over the set **{touching, grasping}** and $\langle noun \rangle$ ranges over the set **{hand, table, cup, ping-pong ball}**. The things in the verb category are binary predicates; the nouns, unary predicates (note, not individuals). The semantics are existential; that is, *ach* (grasping hand cup) means that *some* hand should be grasping *some* cup. The semantics will be made more formal in the section on \mathcal{L}_1 .

This definition should help the designer of each module to determine what its inputs and outputs must be in order to enable the desired top-level competences.

The particular top-level goal of the agent will, initially, be specified at compile time, but it should be a simple extension to allow top-level goals of this kind to be entered by a user at the keyboard.

2.2 Language and Database Definitions

2.2.1 \mathcal{L}_1

This section contains a preliminary specification for the language to be used in specifying goals and internal conditions for the Level 1 demonstration.

2.2.1.1 Abstract \mathcal{L}_1 : \mathcal{AL}_1

First order language over the following symbols:

Constants:	table1, robot-hand1, camera1
Unary predicates:	table, robot-hand, camera, cup, ping-pong ball
Binary predicates:	grasping, touching, in
Unary functions:	weight
Binary functions:	relative pose

Obviously, auxiliary predicates will be required for expressing any interesting conditions and manipulation strategies. E.g., how do we express the fact that the cup is upside down?

$$cup(c) \wedge upside - down(c)$$

or

$$surface(b) \wedge table(t) \wedge cup(c) \wedge bottom - of(b, c) \wedge touching(b, t)$$

2.2.1.2 Concrete \mathcal{L}_1 : \mathcal{CL}_1

To serve as a concrete language for programming, we have to define the Gapps expressions that express conditions abstractly defined by \mathcal{AL}_1 . Gapps has two places where "conditions" or propositions play a role:

1. as the first operand in an *if* expression: (*if condition subgoal1 subgoal2*); and
2. as the operand of an *ach* or a *maint* expression, with the added complexity that the *condition* is factored into a compile-time tag and run-time parameters: (*ach/maint tag-of-condition parameters-of-condition*).

2.2.1.2.1 Testable Conditions The first case is easier to deal with: we can use exactly the first order language \mathcal{AL}_1 , with quantifiers interpreted as finite quantification over indices in the data base (rather than real-world objects—a major pun).

So, we can implement a Rex function, *db-test* which takes a Lisp s-expression that encodes a statement in \mathcal{AL}_1 and returns true, false, or dont-know. It is possible to refer to individuals in these expressions by using a database index, gotten by calling the function *a* with a variable name and an expression in \mathcal{AL}_1 in which that variable occurs free. (For example (*a 'x (cup x)*)). It will be useful to augment \mathcal{AL}_1 with the quantifier $\exists!$; this obviates the need for a the function, as well.

2.2.1.2.2 Goal Conditions The second case is trickier, since the goal reduction rules don't just "evaluate" the condition, they trigger off its syntactic shape. For now, define a subset of goals

$$(ach/maint (\langle binary \rangle \langle unary \rangle \langle unary \rangle) [u1 \ v1 \ v2])$$

where the condition encoded by the ordered pair $\langle (r \ p_1 \ p_2), [] \rangle$ is $\exists x, y. p_1(x) \wedge p_2(y) \wedge r(x, y)$. Far from general, but a way of getting started. (This is the interpretation used in the section describing the external goals for this level.)

2.2.2 Database Definition

Let LP and LR be lattices with elements finitely representable and with operations \sqcap (meet), \sqcup (join), \models (entails). These lattices can be primitive lattices of several types

- finite lattices defined explicitly (e.g., type hierarchies)
- interval lattices with elements $[x, y]$, where

$$[x_1, y_1] \sqcap [x_2, y_2] = [\max(x_1, x_2), \min(y_1, y_2)].$$

- product lattices with elements $[x_1, \dots, x_n]$, where

$$[x_1, \dots, x_n] \sqcap [y_1, \dots, y_n] = [x_1 \sqcap y_1, \dots, x_n \sqcap y_n].$$

Intuitively, the elements of LP represent primitive unary properties that can hold of objects in the robot's world, and the elements of LR represent primitive binary relations. (The non-primitive ones will be computed from the database by database projection.)

In addition to meet, join, etc., we assume we are also given the following operations on lattice elements:

$$\begin{aligned} \text{deg1} &: LP \rightarrow LP \\ \text{deg2} &: LR \rightarrow LR \\ \text{triang} &: LR \times LR \rightarrow LR \end{aligned}$$

Let 0 (1) be the minimal (maximal) elements of some lattice determined by context. Let $DB = [P, R]$, where P is a vector of size n with elements drawn from LP , and R is an $n \times n$ array with elements drawn from LR . Let IN be the input variable, taking as values triples $\langle u, v, w \rangle$ where $u \in LP$ and v, w are each n -tuples of elements of LR . (Intuitively, u represents a unary description of an object; v and w represent the relations between it and the other objects. Most of v and w can be 0, possibly excepting the objects relation to the camera and/or the arm. Obviously, we could have also have a vector of IN s.)

We can define the DB component's next-state function as

$$f(DB, IN) = \text{infer}(\text{insert}(IN, \text{purge}(\text{degrade}(DB))))$$

with the following function definitions.

Degrade:

$$\text{degrade}([P, R]) = [P', R'],$$

for all i ,

$$P'[i] = \text{deg1}(P[i])$$

and for all i, j ,

$$R'[i, j] = \text{deg2}(R[i, j]).$$

Purge:

$$\text{purge}([P, R]) = [P', R']$$

Let m be the index of the least important object in DB; then

$$P'[i] = \begin{cases} 0 & \text{if } i = m \\ P[i] & \text{otherwise} \end{cases}$$

$$R'[i, j] = \begin{cases} 0 & \text{if } i = m \\ 0 & \text{if } j = m \\ R[i, j] & \text{otherwise} \end{cases}$$

Insert:

$$\text{insert}(\langle u, v, w \rangle, [P, R]) = [P', R']$$

Let m be the index of the least important object in DB (the index just purged)

$$P'[i] = \begin{cases} u & \text{if } i = m \\ P[i] & \text{otherwise} \end{cases}$$

$$R'[i, j] = \begin{cases} v[j] & \text{if } i = m \\ w[i] & \text{if } j = m \\ R[i, j] & \text{otherwise} \end{cases}$$

Infer:

$$\text{infer}([P, R]) = \text{infer1}(\text{infer1}(\dots \text{infer1}([P, R]) \dots))$$

and

$$\text{infer1}([P, R]) = \text{propagate}(\text{merge}([P, R]))$$

Merge:

$$\text{merge}([P, R]) = [P', R']$$

Let i, j be the first mergeable pair (i.e., the first pair such that $R[i, j] \models ' = '$). For all k ,

$$P'[i] = P[i] \cap P[j]$$

$$R'[i, k] = R[i, k] \cap R[j, k]$$

$$R'[k, i] = R[k, i] \cap R[k, j]$$

Propagate:

$$\text{propagate}([P, R]) = [P, R']$$

For all k ,

$$R'[i, j] = R[i, j] \cap \text{triang}(R[i, k], R[k, j])$$

2.3 Interface Definitions

2.3.1 Perception to Recognition: Vision tools on Natasha

2.3.1.1 General interface notes

2.3.1.1.1 Port numbers

Natasha system control port: $1024 + 442 = 1466$

Natasha receive port: $1024 + 444 = 1468$

Natasha reply port: $1024 + 445 = 1469$

2.3.1.1.2 Datagram formats To start the system, send a one word datagram to Natasha's system control port containing the number 1. This starts a server which will then handle requests and respond on the receive and reply ports. To stop the system, send a one word datagram to the system control port containing the number 0.

Request datagram packets begin with a tick number, followed by a single code word indicating type of request, followed by additional words specific to the type of request being made.

Reply datagrams will begin with the code word and parameter words from the request being replied to, followed by additional words specific to the type of request.

2.3.1.1.3 Scheduler behavior Once started by the start system message, the sign-correlation scheduler will monitor incoming datagrams on Natasha's receive port. Packets will be read out of the input buffer until the last one is found. All packets with a tick number smaller than that of the last packet (unless packets can come out of order) will be discarded. The most recent packet will be decoded and dispatched to the appropriate measurement tool. The resulting measurement will then be returned to the requester and this cycle will be repeated.

Note that the input buffer seems to have a capacity for about 40 short datagrams. When it is full it ignores new stuff. Thus if too many new requests are sent while Natasha is working on an earlier one, the 41st and later requests will be discarded.

2.3.1.2 Motion Tool

The motion tool will measure motion over its full field of view and report the position, velocity, and approximate size of the n fastest moving regions. (n will be a small number in the range (1 to 4)).

1. Motion request datagram format

Tick number The number of the Rex tick on which this message was generated.

Request code = 100. Code for a motion measurement.

2. Motion reply datagram format

Tick number The number of the Rex tick on which this message was generated.

Request code = 100. Code for a motion measurement.

motion vector 5 word motion cluster.

- (a) x position in mm relative to calibration plane origin
- (b) y position
- (c) x velocity in mm per second relative to calibration plane
- (d) y velocity
- (e) cluster area in square mm at calibration plane

2.3.1.3 Stereo Tool

The stereo tool will make an m by m set of range measurements about a specified location in the visual field—m on the order of 4 or 5. It is capable of taking advice on the expected range to the surface at that location. The tool will return a confidence measure for the measurement, the average range to the surface, the gradient of the surface, its x and y curvature, and the disparity range searched.

1. Stereo request datagram format

Tick number The number of the Rex tick on which this request was generated.

Request code = 200. Code for a stereo measurement.

x position center position of measurement in mm relative to calibration surface origin.

y position center position of measurement in mm relative to calibration surface origin.

z position estimate in mm relative to calibration surface; -32768 means no estimate available.

measurement patch diameter in mm relative to calibration surface.

2. Stereo reply datagram format

Tick number The number of the Rex tick on which this request was generated.

Request code = 200. Code for a stereo measurement.

x position center position of measurement in mm relative to calibration surface origin.

y position center position of measurement in mm relative to calibration surface origin.

z position estimate in mm relative to calibration surface; -32768 means no estimate available.

measurement patch diameter in mm relative to calibration surface.

failure code = if not zero then there was a problem indicated by following code numbers.

1 requested measurement lies outside of active camera field of view.

2 failed to measure a good correlation over the range searched

confidence 0 to 100 scale with 100 the highest confidence in following measurement data.

average range Height above (negative - below) calibration plane in mm. Calibration plane position established at system calibration time.

x gradient of surface In mm per mm on calibration plane.

y gradient of surface In mm per mm on calibration plane.

x curvature of surface In mm per mm per at calibration plane.

y curvature of surface In mm per mm per at calibration plane.

condition codes set bits in this word indicate following conditions (bit zero is least significant).

0 dropouts seen at positive y side of region

1 dropouts seen at negative y side of region

2 dropouts seen at positive x side of region

3 dropouts seen at negative x side of region

2.3.2 Perception to Recognition: Vision tools on Boris (or Wayback)

2.3.2.1 General interface notes

2.3.2.1.1 Port Numbers

Boris Shape tool requests sent to port: boris:(1024 + 450)

Boris Shape tool replies returned to port: boris:(1024 + 451)

Wayback shape tool requests sent to port: wayback:(1024 + 272)

Wayback shape tool replies returned to port: boris:(1024 + 273)

2.3.2.1.2 Scheduling The shape tool will await messages from its assigned port in a sleeping state, permitting other processes on the machine to run. When one or more packets are received, all available packets are read, the latest (largest) tick number is noted, and all packets having a tick number less than this are discarded. The shape tool will then work on the remaining requests in the reverse of the order in which they were received.

The shape tool will sleep for 1ms at intervals to be agreed upon, and between processing packets, in order to let other processes running on the machine obtain a time slice more frequently than the scheduler might otherwise enforce.

2.3.2.1.3 Datagram formats Datagrams to the shape tool will take the form of a request code, and a tick number, followed by a series of parameters according to the following definitions.

```
typedef Coord      int;      /* Pixels */
typedef Coord2     int;      /* Square pixels */
typedef Angle      int;      /* RAUs (4096 per circle) */

typedef enum
{
    shape_status = 0,          /* Trigger a null response. */
    shape_axes = 1,           /* Ask for orientation of major/minor axes */
    shape_parameters = 2,     /* Ask for centre of blob */
    shape_signature = 3,      /* Ask for signature of blob */
    /* Future extensions. */
} ShapeToolRequestCode;

typedef enum
{
    mode_normal = 0, /* Default segmentation mode */
} SegmentationMode;

struct ShapeToolRequestPacket
{
    int      tick;      /* Monotonic tick number. */
    ShapeToolRequestCode code; /* Request code. */
    SegmentationMode mode; /* Unused. */
    Coord    x;         /* Coords of region to look near. */
    Coord    y;
    Coord    w;         /* Size of region to look in. */
    Coord    h;
};

typedef enum
{
    shape_status_null = 0, /* Empty response to any query */
    shape_status_blank = 1, /* Found no shape to analyse */
    shape_status_ok = 2, /* Found shape to analyse */
    shape_status_truncated = 3, /* Shape may extend beyond portion analysed. */
} ShapeToolStatusCode;

struct ShapeToolResponsePacket
{
    int      tick;      /* Tick number from request. */
    ShapeToolRequestCode code; /* Request code. */
    SegmentationMode mode; /* Unused. */
    Coord    x;         /* Coords of region to look near. */
    Coord    y;
```

```

Coord          w;          /* Size of region to look in. */
Coord          h;

ShapeToolStatusCode  status;
unsigned           trunc;  /* Unused */

union
{
    /* Response to shape_parameters */
    struct
    {
        Coord          cent_x; /* Centre of area */
        Coord          cent_y;
        Coord2         area;    /* Area */
        Coord          perim;   /* perimeter length */
        Coord          major;   /* Length of major axis */
        Coord          minor;   /* Length of minor axis */
        Angle          orient; /* of major axis ccw from y=0 */
    } shape_parameters;

    /* Response to shape_signature */
    struct
    {
        Coord          x;      /* Position of peak */
        Coord          y;
        int            strength; /* Height of peak */
    } signature[SignatureCount]; /* Report up to SignatureCount peaks */
} response;
};

```

2.3.2.2 Shape tool

The shape tool will analyse a rectangular region of the image centered at x , y , with size $2*w \times 2*h$. The region will be segmented according to the Segmentation mode field: currently only a default mode is defined.

- A `shape_status` request will cause a packet with `status_null` to be returned.
- If no region is found, the status parameter will be set to `status_blank`.
- If there is a region, the status parameter will be set to `status_ok`. If there is more than one region in the field, the report will describe one of the regions. This might be the largest region entirely enclosed within the field.
- If the region reported upon has pixels in the edgemost row or column of the field, it cannot be distinguished that the region does not extend beyond the field, and the status parameter will be set to `status_truncated`. A future extension might use the `trunc` field to report which edges or corners of the field cut across the region.

2.3.2.2.1 Request shape parameters This request will ask for the centre position, area, perimeter, and orientation and length of major and minor axes of the blob in the region.

2.3.2.2.2 Request shape signature This request will ask for the raw signature of the region shape: this will be the coordinates of the **SignatureCount** highest peaks on the region, along with their heights. The coordinates might be object centred, and the heights might be normalized so that the largest has a specified value.

2.3.3 Recognition to Database

The interface between the database and recognition components will consist of an array of records. Each record represents information about an object, including a set of unary properties, such as type, and the relation of the object to each of the landmark objects. The size of the array will be 2 initially, but may grow as recognition gets to be more useful.

Landmark objects are things that are unique and known to all of the components. We propose the following set of landmark objects: **gripper**, **camera**, and **table**. The table's frame will be the same as the frame of the robot, which is centered at the base. This is necessary because the dead-reckoning capabilities of the arm continuously report the location of the gripper in the coordinates of the arm-base. Thus, if we know the location of, say, a ping-pong ball with respect to the gripper and the location of the gripper with respect to the base, we can deduce the location of the ping-pong ball with respect to the arm base. In the next tick, imagine that the gripper moves with respect to the base, but the ping-pong ball does not. In this case, we will have new information about the relative position of the gripper and the base; when we combine that information with the relative position of the ping-pong ball and base, we will have current information about the relative position of the ping-pong ball and the gripper.

The gripper also has a standard frame of reference associated with it. The camera frame will be centered between the camera centers and pointing down the middle (the average of the directions of the cameras).

In order to make the data structures uniform, we will define three types: **property**, **relation**, and **recognition-data**. These types may be the same as the ones that are used internally in the database (that would simplify things), but need not necessarily be the same.

The **property** data type will have the following fields:

type The type of the object will be one of primitive types or a disjunction of a set of them. The primitive types will be: **gripper**, **camera**, **table**, **pp-ball**, and **cup**.

gripper-open Is the object open? Can only be true of the gripper. Needs no valid bit because we always know whether or not it is open. It is sort of disgusting to have to have this in every entry, but Rex requires fixed data types. It's possible that we could use this field to mean different things for different types, if we ever need to.

The **relation** data type will have the following fields:

relative-pose Bounds on the position and orientation of the first object in the second object's frame of reference. Jeff is working on exactly how these bounds are to be represented.

relative-velocity Bounds on the velocity of the first object in the frame of the second object.

relative-force Bounds on the force exerted by the first object on the second. The gripper will almost always be one of the participants in this relation when it is informationful. If we want to say more useful things, like what part of the object is exerting force on the gripper, this field will have to get more complicated.

Finally, the **recognition-data** data type will have the following fields:

valid A Boolean value indicating whether the remainder of the record is valid. This allows the recognition component to not fill up the entire set of records if it doesn't have interesting data to report.

object-properties A value of type **property** giving the unary properties of this object.

gripper-rel A value of type **relation** giving the relation between the object and the gripper. Order is important. We are describing the object in the gripper's frame. If the relations are invertible, that's all we need specify. If they are not, we will have to specify the relation between the gripper and the object, as well. For now, assume that they are inverses.

camera-rel A value of type **relation** giving the relation between the object and the camera.

table-rel A value of type **relation** giving the relation between the object and the table.

2.3.4 Database to Database Projection

On each tick the database component will output its entire contents to the database projection component. The database will be a Rex module, so its output will be in list form. Specifically, it will be a list of Boolean and floating-point values encoding the various properties of and relations between the objects in the database.

The beginning of the output list will correspond to the vector in the database, each cell of which stores characteristics of a single object (type and mass, for instance). An object's type will be encoded by a vector of Booleans whose indices correspond to the possible types. A Boolean in the vector will be set to true if the object might be of the type associated with its index. For example, if the object is known to be a cup, only the cup Boolean will be true, and if the object is known to be either a cup or a gripper, the Booleans for cup and gripper will be true. For the first demo, each cell of the database vector will contain a vector of five Booleans that encode an object's type and one Boolean indicating whether the gripper is open (only relevant for grippers). There will be five cells in the vector to begin with, so the first section of the database's output will be 30 Boolean values.

The remainder of the output list will correspond to the two-dimensional array in the database which stores the relations between objects. For the first demo, each cell $[a, b]$ of the array will contain twelve floating-point values which describe the position of object a in b 's frame of reference. There will be ten useful cells in this array at first (one for each possible pair of objects), so this part of the output list will have 120 floating-point values.

The database projection component will read the values from the database's output list into data structures with the same form as those in the database. The information stored in these structures will then be used to answer the action component's queries about the objects.

2.3.5 Database Projection to Goal Strategies

For the Level 1 demonstration, the interface between the arm control action component and the database will be a language based on a simple epistemic logic. Expressions in the language will evaluate to Rex circuitry that extracts the truth value of desired conditions or object indices from the database. In addition, there will be a set of functions that extract parametric information about database objects from their indices.

2.3.5.1 Database Language

The database language will consist of a small set of Rex functions and a separate language, used by these functions, that can generate circuitry that accesses the database.

Types: The database language uses two special types.

1. *objs* are indices into the database that correspond to some database object.
2. *k-conds* or knowledge conditions represent the system's knowledge about a particular condition. They can have three values, $\{1, 0, \perp\}$, that roughly correspond to 'known to be true', 'known to be false', and 'unknown'.

Terms: Any Rex expression of type *obj* is a term. There is only one primitive function that returns terms.

1. $(a \ \bar{x} \ k\text{-exp})$ returns an *obj* or list of *objs*. \bar{X} is an atom or list of atoms, and *k-exp* is a knowledge expression as described below. Intuitively a returns an *obj* or a set of *objs* from the database that satisfy some condition. It finds the set by sequentially binding the atoms in \bar{x} to the *objs* in the database. a is equivalent to a .

Condition Expressions: Condition expressions are of type *k-cond*. They can only appear as the argument to the *know* function, described below, or as part of other condition expressions. The bottom level condition expressions are a set of conditions that can be directly tested on the database. There will be one domain independent condition.

1. (*equal term term*) tests whether the two terms reference the same database object, i.e. are the indices equal.

The following is a list of conditions that we expect to implement for the goals of grasping and touching.

1. (*grasping term term*) tests whether the first object is grasping the second object. For the current demo this test can only be 1 or \perp if the first object is the hand; if the first object is anything else, it must be 0.
2. (*touching term term*) tests whether the first object and the second object are touching.
3. (*can-grasp term term*) tests whether there is sufficient space around the second object for the first object to grasp it. This condition has the same constraints as *grasping*.
4. (*can-touch term term*) tests whether there is sufficient space around the second object for the first object to touch it.
5. (*in term term*) tests whether the first object is entirely within the bounds of the second object.
6. (*clear-between term term*) tests whether there is sufficient space between the objects such that the first object can move so it is touching the second object.
7. (*fully-above term term*) tests whether the lowest Z coordinate of the first object is above the highest coordinate of the second object.
8. (*above term term*) tests whether the lowest Z coordinate of the first object is above the lowest Z coordinate of the second object.
9. (*xy-bounds-intersect term term*) tests whether the projections of the two objects onto the table intersect.
10. (*table term*) tests whether the object is a table.
11. (*robot-hand term*) tests whether the object is a robot hand.
12. (*human-hand term*) tests whether the object is a human hand.
13. (*hand term*) tests whether the object is a hand.
14. (*pp-ball term*) tests whether the object is a ping-pong ball.
15. (*cup term*) tests whether the object is a cup.
16. (*soda-can term*) tests whether the object is a soda can.
17. (*freezer-container term*) tests whether the object is a freezer-container.
18. (*container term*) tests whether the object is one of *cup*, *soda-can* or *freezer-container*.
19. (*open term*) tests whether the object is open. Can only be known if the object is a container.

20. (*gripper-open term*) test if the object's gripper is open. Can only be known if the object is a hand.
21. (*moveable term*) tests whether a given object can potentially move. It is determined just from the type of the object and hence does not take into consideration whether there is anything currently blocking the object from moving.
22. (*has-opening term*) tests whether a given object has an opening. This condition is also determined from just the type of the object. Can only be known if the object is a container.

Complex Conditions : More complex conditions can be formed by combining other conditions using the following operators:

1. (*not pred-exp*) maps 1 into 0, 0 into 1 and \perp into \perp .
2. (*and pred-exp pred-exp*) returns 1 when both arguments are 1, 0 when at least one argument is 0, and \perp otherwise.
3. (*or pred-exp pred-exp*) returns 0 if both arguments are 0, 1 if at least one argument is 1, and \perp otherwise.
4. (*exists \bar{x} pred-exp*) returns 1 if there is an *obj* for each atom of \bar{x} for which *pred-exp* is 1, returns 0 when *pred-exp* is 0 for all bindings of \bar{x} , and \perp otherwise.
5. (*exists-unique \bar{x} pred-exp*) returns 1 if there is exactly one binding of \bar{x} for which *pred-exp* is 1, 0 if *pred-exp* is 0 for all bindings of \bar{x} or *pred-exp* is 1 for more than one binding of \bar{x} , and \perp otherwise.
6. (*for-all \bar{x} pred-exp*) returns 1 if for all bindings of \bar{x} the value of *pred-exp* is 1, 0 if there is some binding of \bar{x} for which *pred-exp* is 0, and \perp otherwise.

Knowledge Expressions: Knowledge expressions are Boolean-valued Rex expressions that use only the operators *andm*, *orm*, *notm*, and *know*. The first three of these are just the standard Rex functions.

1. (*know pred-exp*) evaluates *pred-exp* and then maps 1 to 1b, 0 to 0b, and \perp to 0b. Intuitively it tests whether *pred-exp* is 'known to be true'.

2.3.5.2 And Now More Formally

$\langle \text{extended-rex} \rangle \rightarrow \langle \text{knowledge-expression} \rangle \mid \langle \text{term} \rangle \mid \langle \text{rex-expression} \rangle$

$\langle \text{knowledge-expression} \rangle \rightarrow$
 $\langle \text{andm} \langle \text{knowledge-expression} \rangle \langle \text{knowledge-expression} \rangle \rangle$
 $\mid \langle \text{orm} \langle \text{knowledge-expression} \rangle \langle \text{knowledge-expression} \rangle \rangle$
 $\mid \langle \text{notm} \langle \text{knowledge-expression} \rangle \rangle$
 $\mid \langle \text{know} \langle \text{condition-expression} \rangle \rangle$

$\langle \text{condition-expression} \rangle \rightarrow$

```

(and <condition-expression> <condition-expression>)
| (or <condition-expression> <condition-expression>)
| (not <condition-expression>)
| (exists  $\bar{x}$  <condition-expression>)
| (exists-unique  $\bar{x}$  <condition-expression>)
| (forall  $\bar{x}$  <condition-expression>)
| (equal <term> <term>)
| (<ground-condition> <extended-rex >* <term>+)

```

```

<term> →
  (a  $\bar{x}$  <knowledge-expression>)
  | <rexobj>

```

where \bar{x} is an atom or list of atoms, <ground-condition>s are a set of Rex functions of type *k-cond*, and <rex_{obj}>s are Rex expressions of type *obj*.

2.3.5.3 Other functions

The above functions only return *k-conds* and *objs*. There will also be a set of Rex functions that extract parametric values about an *obj* from the database. The following are functions useful for touching and grasping.

1. (grasping-points *obj*) returns three object relative poses at which the object can be grasped.
2. (opening *obj*) returns the bounds on the radius of the given object's opening and the vector from the center of the object to the center of the opening. If (has-opening *obj*) is false then the function is undefined.

2.3.5.4 Examples

1. An object that is known to be held by a hand.

```
(an object (know (grasping (an obj2 (hand obj2)) object)))
```

2. The object with the greatest highest Z coordinate excluding the hand and any object the hand is holding.

```

(an object (know (for-all obj2
                    (or (above object obj2)
                        (grasping (an obj3 (know (hand
object))))
                    obj2))))

```

3. The object with the greatest highest Z coordinate that is above the given object and whose X and Y bounds intersect with the given object.

```
(defun highest-object-above (object)
  (an obj2 (andm (know (XY-bounds-intersect object obj2))
                 (know (above obj2 object))
                 (notm (know (exists obj3 (above obj3 obj2)))))))
```

4. The object that the given object is contained within.

```
(defun surrounding-object (object)
  (an obj2 (know (in object obj2))))
```

2.3.6 Goal Strategies to Manipulation

The consensus is that these two components are too intertwined to make it useful to specify an interface between them in advance. As the demo tasks become more complex, this division may be more useful.

2.3.7 Manipulation to Arm Control

2.3.7.1 Overview

REX can talk to the ZERO arm over the Ethernet via pseudo-UDP packets. A command packet contains a few bytes of command information followed by some command parameters. They are fixed length packets.

Once a packet is received by the ZERO controller, it begins execution of the command (if possible), and sends back a status packet. The status packet is parallel to the command packet in that it has a few bytes of status information followed by the arm data.

2.3.7.2 Packet Descriptions

2.3.7.2.1 Command Packet The command packet consists of 53 bytes described in the table below:

Byte #	Description
0	First header byte - arbitrarily set to the char 'A'
1	Second header byte - set to the char 'B'
2	Command byte - describes one of the N possible commands
3	Output byte - commands digital devices to be on or off
	Bit # Device / State
	0 Hand - 0 = closed, 1 = open
	1-7 Unused
4-27	Commanded hand position given as a 6x1 XYZ/Angle-Axis vector or as a 6x1 joint angle vector (P1 - P6): 6 numbers encoded as 4 byte floats, LSB first. UNIX and the PC use the same IEEE standard floats, but the SUNs store floats MSB first whereas the PC, the DEC machine and the lisp machine all store floats LSB first. The network standard is to send floats MSB first, but we are sending the LSB first, anyway.

- 28-51 Force command parameters (F1 - F6): 3 forces along the XYZ gripper axes and 3 torques about the XYZ gripper axes encoded as 4 byte floats, LSB first.
- 52 Bytewise checksum - { [sum(bytes 0-51)] modulo 256 }

The first two bytes and the last byte are not really used for anything - they are left over from the serial communication days. We leave them in just in case, we have to go back to serial communication.

2.3.7.2.2 Status Packet The status packet also consists of 53 bytes which are described in the table below:

Byte #	Description
0	First header byte - arbitrarily set to the char 'A'
1	Second header byte - set to the character 'B'
2	Status byte with the bit definitions:
	Bit # Description
	0&1 Command status:
	0 = last command accepted and completed
	1 = execution of last command in progress
	2 = last command failed for some reason
	3 = bad command number in last packet
2	1 = Bad checksum (not currently used), 0 = OK
3	1 = Arm is in motion, 0 = Arm stationary
4	1 = Last move command was aborted due to excessive forces
	0 = OK
5	1 = Command parameters out of range, 0 = OK
6	1 = Arm not homed, 0 = Arm ready to go
7	1 = Gripper open, 0 = gripper closed
3	Input byte - input from digital sensors
	Bit # Device / State
	0-7 Unused
4-27	Arm joint angles - Joint angles 1 - 6 in radians encoded as 4 byte floats, LSB first.
28-51	3 forces along the XYZ gripper axes and 3 torques about the XYZ gripper axes encoded as 4 byte floats, LSB first.
52	Bytewise checksum - [sum(bytes 0-51)] modulo 256

Again, the header bytes and the checksum are not used, but are left for reverting to the serial communications.

2.3.7.3 Receiving and Executing Commands

The general flow of operation is as follows: The ZERO controller waits for an ethernet packet to arrive. When any packet arrives, it assumes it is an arm command, and begins processing the command, setting the appropriate status flags as needed.

Some commands are initiated, but will not be completed before the next command arrives. This situation is indicated by the first two status bits in the status byte. Most commands, however, will be completed in the initial processing.

After a command is processed, a status packet is returned which reflects the outcome of that commands' processing. The controller then waits for the next command packet. Commands are double buffered as they come in over the ethernet. When ready to process a new command, it is the most recent command received which is copied into a working buffer for processing. If command packets come in too fast (at more than about 15 hz), some of them will be dropped. There is currently no indication that a packet has been dropped.

2.3.7.4 Command Descriptions

The commands encoded by the command byte fall into roughly the two categories of setting control parameters, or starting or modifying some arm motion. These commands are accompanied by parameters passed in the position and force command vectors in the rest of the command packet.

In specifying the arm configuration, the position vector must represent the position of the hand in the world in some particular set of coordinates. For some commands, arm positions are commanded as Cartesian coordinates specifying the position and orientation of the gripper relative to a base frame which is fixed relative to the table.

The gripper frame is encoded as a 6x1 position / angle-axis vector. The first three elements of the vector are the X , Y , and Z coordinates of the origin of the gripper relative to the base frame in millimeters. The second three elements represent a vector in the base frame, such that if we rotate the base frame about this vector, we will get the same orientation as the gripper frame. The amount that we need to rotate about this vector is given by its length in radians.

Algorithms for converting back and forth between rotation matrices and angle-axis vectors are given in John Craig's robotics book in Chapter 2. One special case which he leaves as an exercise to the reader is for converting to the angle-axis form when the rotation angle is 180 degrees. If the rotation matrix is given as R , the rotation vector for 180 degree case is simply the sum of the columns of the matrix $(R + I)$. This vector should be scaled to have length π .

Other arm commands specify the configuration of the arm directly in joint angles. Note that the position vector returned in the status packet is always in joint angle coordinates. The forward kinematic function of the arm must be invoked to convert the joint angles into a Cartesian frame.

All of the commands available to in the REXARM program are detailed below:

2.3.7.4.1 Parameter Setting Commands

0 NOP - good for waiting for something to happen.

1 Robot initialization - turns the servos on and declares that the arm is in the home position. Should only be used in homing operations.

- 2 Robot reset - turns the servos off and forgets where the arm is.
- 5 Zero force sensor - declare a new zero position for the force sensor for force thresholded moves.
- 7 Set zero position - tells the arm what its joint angles are when it is positioned in the nest. Use only in homing the robot. Home position is specified in joint angles in the position command vector.
- 8 Set speed - a single joint speed parameter between 0.0 and 1.0 stored in P_1 , the first element of the position vector. Smaller is slower, bigger is faster.
- 9 Set acceleration - a single joint acceleration parameter between 0.0 and 1.0 stored in P_1 , the first element of the position vector. Smaller is slower, bigger is faster.
- 15 Float - turn the motors off while continuing to monitor joint positions. Currently, the correct position is not returned in the status packet until a move command is issued. A freeze command should be issued before issuing any move commands.
- 16 Freeze - turn the motors on, servoing to the arm's current position.
- 127 Halt execution of the program. The arm will automatically go back home.

2.3.7.4.2 Fixed position commands These move commands are initialized, but will not be completed for several ticks. If one of these commands is issued, communication will cease until the move is finished. These commands are not generally useful for REX style programming but will work as long as the REX program is content to live with old data until the move is finished.

- 10 Joint space move - move the arm to the joint angles specified in $P_1 - P_6$.
- 11 Relative joint space move - increment the arm joint angles by the amounts specified in $P_1 - P_6$.
- 12 Single joint move - moves the joint 1-6 (specified as 1.0 - 6.0 in P_1) to a specified angle (in degrees in P_2)
- 17 Add wobble - sets the magnitude of a wobble to be superimposed on the three wrist joints for all subsequent fixed position move commands. Magnitude in degrees is specified in $P_1 - P_3$.
- 18 Turns the wobble off.
- 19 Add joint space via point - specify the next point in a continuous path in joint coordinates $P_1 - P_6$.
- 20 Translate gripper - translate the gripper by the X, Y, Z increments specified in $P_1 - P_3$ without changing the orientation. Coordinates are in millimeters in base frame coordinates.

- 21 Hand relative translation - Same as command 20, but the coordinates are specified relative to the hand coordinate frame.
- 22 Hand rotation - rotates the hand about the axis specified by $P_1 - P_3$ (in base coordinates) by the angle specified by P_4 (in degrees). The fingertips will not translate.
- 23 Hand relative rotation - same as command 22, but the axis is specified in hand coordinates.
- 25 Cartesian move - move the gripper to the Cartesian coordinates $P_1 - P_6$ encoded as an XYZ/angle-axis vector.
- 26 Add Cartesian via point - same as command 19, but the via point is specified with an XYZ/angle-axis vector.
- 27 Run path - executes the continuous path set by the previously specified via points.

2.3.7.4.3 Variable Position Moves These move commands specify a single goal position which can be modified continuously. When one of these commands is issued, the arm will start moving towards the goal point, the controller will return a status packet and then look immediately for the next command. New goal positions can be sent whether the arm has reached the goal, or not. These are the commands typically used by REX programs.

- 28 Cartesian move - move the gripper to the Cartesian coordinates $P_1 - P_6$ encoded as an XYZ/angle-axis vector. A command 29 should be issued before giving any other kind of move command.
- 29 Halt move - take the arm out of variable position move mode and servo to the current position of the arm.

One last detail in the command section is the output byte. This is used to turn on or off binary devices. The only device currently used is the gripper, which is commanded by bit 0.

2.3.7.5 Ethernet, IP and UDP Particulars

The PC communicates over the ethernet using a 3COM 503 ethernet adapter and the KA9Q public domain device driver. This device driver sends and receives raw ethernet packets. Unfortunately, the rest of the REX related machinery insists upon using several layers of communication protocols above the ethernet level. Communication takes place through UDP packets, which are then bundled as IP packets, which are then bundled as ethernet packets.

REXARM uses none of the features of the IP or UDP protocols. If it receives a packet with its ethernet address, it strips away all of the header information, assuming the packet is an arm command.

Sending status packets back to the REX machine, however, is difficult, because the packets must contain IP and UDP headers which are realistic enough to fool the UNIX communication software.

The outer layer of headers is the ethernet header containing the ether destination address, ether source address, and type code (here, we use 0x0800 to indicate that the next lower layer is an IP packet). An ethernet checksum is tacked onto the end of the ethernet packet by the ethernet hardware.

The IP header starts off with version, IHL and type of service information, all copied verbatim from other similar packets on the net. (I don't know what these things mean, but simply hardwiring their values seems to work for now.) Next is the IP packet length and identification. The packet length is the length of the IP packet in bytes, and the identification we arbitrarily set to the number of packets sent. The flags, fragment offset, and time to live fields are again just hardwired to similar values seen on the net. The protocol is set to 0x11 for UDP packets, and the IP header checksum is calculated through some completely convoluted means. The last things to deal with are the IP source and destination addresses.

The next level header, the UDP header, is quite a bit simpler - it contains the UDP source and destination port numbers, the UDP packet length, and a UDP checksum. The UDP checksum is simply set to 0, but nobody seems to care.

The ethernet addresses, IP addresses and UDP port numbers of all the relevant machines are listed below:

```
zero_enet_addr[] = { 0x02, 0x60, 0x8c, 0x0c, 0x15, 0x3a };
```

```
zero_ip_addr[] = { 0xc0, 0x2a, 0x08, 0x16 };
```

```
sherman_enet_addr[] = { 0x08, 0x00, 0x20, 0x01, 0x22, 0xce };
```

```
sherman_ip_addr[] = { 0xc0, 0x2a, 0x08, 0x02 };
```

```
sherman_udp_port[] = { 0x06, 0x40 };
```

```
wayback_enet_addr[] = { 0x08, 0x00, 0x20, 0x01, 0x01, 0x2e };
```

```
wayback_ip_addr[] = { 0xc0, 0x2a, 0x08, 0x04 };
```

```
wayback_udp_port[] = { 0x05, 0xc2 };
```

```
boris_enet_addr[] = { 0x08, 0x00, 0x2B, 0x0F, 0x0E, 0x41 };
```

```
boris_ip_addr[] = { 0xc0, 0x2a, 0x08, 0x17 };
```

```
boris_udp_port[] = { 0x06, 0x40 };
```

2.3.7.6 Owning and Operating REXARM

The first thing to remember for safe robot operation is to turn the power on only when REXARM prompts you, and turn the power off whenever things look dicey.

Before turning on the PC, first make sure that the two 40 pin and one 10 pin flat ribbon cables are plugged in, as well as the power connector and the air hoses. All these connections are made in the metal box next to the arm. Open up the valve on the air tank to make sure there is air for the gripper. The gauge should read between 40 and 60 psi. Fill up the tank if needed. Make sure the arm is placed in its nest.

With everything in place, turn on the PC and move to the TELEOS directory. Once there, type REXARM2 to start up the current version of the program. If you just want to test a REX program without actually moving the arm, indicate so at the prompt. When asked for the name of the REX machine type 'boris' or 'wayback', depending on which machine is being used. Finally, when prompted, push the green power button on the arm power supply and turn on the air. (Don't bother if you indicated that you don't want to use the arm.) From here on out, it is a good idea to hold onto the remote kill button.

After hitting a key to acknowledge that power is enabled, the arm will go through its homing procedure. The arm will then move to the ready position, and is ready to accept ethernet packets. As commands are sent, packet information will be printed on the screen. Note, however, that while a variable position move is in progress, nothing will be printed to the screen until a command 29 is issued.

At any time, a 'q' typed on the keyboard will halt the arm, send it back to its home position, and terminate the program. The same thing happens when a 127 command is received. After the program halts, turn off the power supply with the red button or the remote switch. Don't forget to turn off the air valve.

2.3.7.7 Known Bugs, Glitches and Deficiencies

1. Robot reset and initialization functions do not set and reset the arm homed flag.
2. Communication halts during fixed position commands. These commands should be made interruptible.
3. Current positions and forces are not returned while floating.
4. There is no prompt for turning off the power.
5. After the arm goes back into the nest, periodically a math floating point error shows up. If this is the case, turn off the arm power and reboot the PC for good measure.
6. There is no function for setting force thresholds on moves.
7. If a variable position command is aborted, another move command must be issued to detect the abort condition. A command 29 will wipe out that information.
8. There is no function for changing the gpper frame.

REPORT OF INVENTIONS AND SUBCONTRACTS

(Pursuant to "Patent Rights" clause) (See Instructions on Reverse Side.)

Form Approved
OMB No. 0704-0240
Expires Sep 30, 1988

1a. NAME OF CONTRACTOR / SUBCONTRACTOR Teleos Research	2a. NAME OF GOVERNMENT PRIME CONTRACTOR AFOSR	3. TYPE OF REPORT (X one) a. INTERIM <input checked="" type="checkbox"/> b. FINAL <input type="checkbox"/>
b. ADDRESS (Include ZIP Code) 576 Middlefield Road Palo Alto, CA 94301	c. CONTRACT NUMBER F49620-89-C-0055	4. REPORTING PERIOD (YYMMDD) a. FROM 890401 b. TO 901109
d. AWARD DATE (YYMMDD) 890401	e. AWARD DATE (YYMMDD) 890401	

SECTION I - SUBJECT INVENTIONS

5. "SUBJECT INVENTIONS" REQUIRED TO BE REPORTED BY CONTRACTOR / SUBCONTRACTOR (If "None," so state)		d. ELECTION TO FILE PATENT APPLICATIONS		e. CONFIRMATORY INSTRUMENT OR ASSIGNMENT FORWARDED TO CONTRACTING OFFICER	
a. NAMES OF INVENTOR(S) (Last, First, MI)	b. TITLE OF INVENTION(S)	c. DISCLOSURE NO. PATENT APPLICATION SERIAL NO OR PATENT NO		(2) Foreign	
		(1) United States	(2) Yes (a) No (b) No	(1) Yes	(2) No
NONE	NONE	N	O	N	E

6. EMPLOYER OF INVENTOR(S) NOT EMPLOYED BY CONTRACTOR / SUBCONTRACTOR		9. ELECTED FOREIGN COUNTRIES IN WHICH A PATENT APPLICATION WILL BE FILED	
(1) (a) Name of Inventor (Last, First, MI)	(2) (a) Name of Inventor (Last, First, MI)	(1) Title of Invention	(2) Foreign Countries of Patent Application
(b) Name of Employer	NONE	NONE	
(c) Address of Employer (Include ZIP Code)			

SECTION II - SUBCONTRACTS (Containing a "Patent Rights" clause)

6. SUBCONTRACTS AWARDED BY CONTRACTOR / SUBCONTRACTOR (If "None," so state)		d. DEAR "PATENT RIGHTS"		e. DESCRIPTION OF WORK TO BE PERFORMED UNDER SUBCONTRACT(S)		f. SUBCONTRACT DATES (YYMMDD)	
a. NAME OF SUBCONTRACTOR(S)	b. ADDRESS (Include ZIP Code)	c. SUBCONTRACT NO (S)	(1) Clause Number	(2) Date (YYMM)	(1) Award	(2) Estimated Completion	
NONE		NONE			NONE		

SECTION III - CERTIFICATION

7. CERTIFICATION OF REPORT BY CONTRACTOR / SUBCONTRACTOR		(Not required if <input checked="" type="checkbox"/> Small Business or <input type="checkbox"/> Non-Profit organization) (X appropriate box)	
a. NAME OF AUTHORIZED CONTRACTOR / SUBCONTRACTOR OFFICIAL (Last, First, MI)	c. I certify that the reporting party has procedures for prompt identification and timely disclosure of "Subject Inventions," that such procedures have been followed and that all "Subject Inventions" have been reported.		
b. TITLE	d. SIGNATURE		e. DATE SIGNED

DD FORM 882 INSTRUCTIONS

GENERAL

This form is for use in submitting INTERIM and FINAL invention reports to the Contracting Officer and for use in the prompt notification of the award of subcontracts containing a "Patent Rights" clause. If the form does not afford sufficient space, multiple forms may be used or plan sheets of paper with proper identification of information by Item Number may be attached.

An INTERIM report is due at least every 12 months from the date of contract award and shall include (a) a listing of "Subject Inventions" during the reporting period, (b) a certification of compliance with required invention identification and disclosure procedures together with a certification of reporting of all "Subject Inventions," and (c) any required information not previously reported on subcontracts awarded during the reporting period and containing a "Patent Rights" clause.

A FINAL report is due within 6 months if contractor is a small business firm or domestic nonprofit organization and within 3 months for all others after completion of the contract work and shall include (a) a listing of all "Subject Inventions" required by the contract to be reported, and (b) any required information not previously reported on subcontracts awarded during the course of or under the contract and containing a "Patent Rights" clause.

While the form may be used for simultaneously reporting inventions and subcontracts, it may also be used for reporting, promptly after award, subcontracts containing a "Patent Rights" clause.

Dates shall be entered where indicated in certain items on this form and shall be entered in four or six digit numbers in the order of year and month (YYMM) or year, month and day (YYMMDD). Example: April 1986 should be entered as 8604 and April 15, 1986 should be entered as 860415.

Item 1a. Self-explanatory.

Item 1b. Self-explanatory.

Item 1c. If "same" as item 2c, so state.

Item 1d. Self-explanatory.

Item 2a. If "same" as item 1a, so state.

Item 2b. Self-explanatory.

Item 2c. Procurement Instrument Identification (PII) number of contract (DFAR 4.7003).

Item 2d thru 5e. Self-explanatory.

Item 5f. The name and address of the employer of each inventor not employed by the contractor or subcontractor is needed because the Government's rights in a reported invention may not be determined solely by the terms of the "Patent Rights" clause in the contract.

Example 1: If an invention is made by a Government employee assigned to work with a contractor, the Government rights in such an invention will be determined under Executive Order 10096.

Example 2: If an invention is made under a contract by joint inventors and one of the inventors is a Government employee, the Government's rights in such an inventor's interest in the invention will also be determined under Executive Order 10096, except where the contractor is a small business or nonprofit organization, in which case the provisions of Section 202 (e) of P.L. 96-517 will apply.

Item 5g (1). Self-explanatory.

Item 5g (2). Self-explanatory with the exception that the contractor or subcontractor shall indicate, if known at the time of this report, whether applications will be filed under either the Patent Cooperation Treaty (PCT) or the European Patent Convention (EPC). If such is known, the letters PCT or EPC shall be entered after each listed country.

Item 6a. Self-explanatory.

Item 6b. Self-explanatory.

Item 6c. Self-explanatory.

Item 6d. Patents Rights Clauses are located in FAR 52.227.

Item 6e thru 7b. Self-explanatory.

Item 7c. Certification not required by small business firms and domestic nonprofit organizations.